MASTER EN SCIENCES INFORMATIQUES

Outils graphiques pour environnements logiciels dirigés par la syntaxe

Paquet, Christophe; Paring, Marc

Award date:
1988

Awarding institution:
Universite de Namur

Outils graphiques pour environnements logiciels dirigés par la syntaxe

(volume 1)

Christophe Paquet et Marc Paring

Promoteur
Monsieur **Axel van LAMSWEERDE**

Mémoire présenté pour l'obtention du grade
de Licencié et Maître en Informatique
par

Christophe PAQUET et **Marc PARING**

Année académique 1987 - 1988

Outils graphiques pour environnements logiciels dirigés par la syntaxe

Résumé

L'obtention de spécifications fonctionnelles est actuellement reconnue comme une étape clé dans le développement d'un système informatique. Le projet SACSO vise à développer un environnement intégré d'outils aidant le spécifieur dans cette étape. D'autre part, à l'heure actuelle, la qualité des interfaces utilisateurs conditionne le succès d'un logiciel.
Ce travail propose, en vue d'améliorer l'interface usager du système SACSO, une boîte à outils réutilisable pour la création d'interfaces usager conviviales. Il présente également une proposition de solution pour la conception d'un outil générique de visualisation d'objets formels, représentés de manière interne sous forme d'arbre de syntaxe abstraite. L'intégration de cet outil dans le système SACSO permettra d'obtenir une représentation graphique d'une spécification construit à l'aide de SACSO.

Abstract

Writing requirements specifications is now widely recognized as a very sensitive step during system developement. The SACSO project aims at building an integrated requirements specification environment to help the specifier in this crucial step. On the other hand, the user-friendliness of a system determines the success of a software.
This dissertation presents a reusable toolbox for creating user-friendly interfaces. This toolbox is used to improve the SACSO user interface. This work also presents a solution for the design of a generic tool allowing the visualisation of formal objects represented internally by an abstract syntax tree. The integration of this tool in the SACSO system will allow a graphical representation of a specification constructed with this system.

En terminant ce mémoire, nous tenons à exprimer notre gratitude aux personnes qui nous ont permis de le mener à bien :

Monsieur A. van Lamsweerde, qui a accepté de diriger ce travail. Ses conseils nous ont été d'une grande aide,

Jeannine Souqières, Nicole Lévy, Jean-Pierre Finance pour l'accueil chaleureux qu'ils nous ont réservé dans l'équipe SACSO,

Alain Lesdalons, Monique De Sylvestri, Roland Lambert et tous les membres du CRIN pour l'ambiance de travail sympathique,

enfin, toutes les personnes qui ont contribué à la réalisation de ce travail .

Christophe Paquet et Marc Paring

TABLE DES MATIERES

Chapitre 4 Modification du multi-fenêtrage de SACSO

Partie III UN OUTIL GENERIQUE DE VISUALISATION GRAPHIQUE D'OBJETS FORMELS

Chapitre 5 Présentation du problème

Chapitre 6 Proposition de solution

INTRODUCTION

Introduction

Dans le cycle de vie de développement d'un système informatique, l'étape généralement difficile et cruciale est celle où s'expriment les demandeurs quant aux fonctionnalités attendues du futur système.
De nombreux outils ont été proposés ces dernières années en vue d'assister l'informaticien dans cette étape de spécification des besoins. Ces outils fournissent généralement au spécifieur les moyens d'exprimer des modèles de spécification permettant de saisir les concepts significatifs du système à développer. De tels outils communément appelés langages de spécification, ne représentent qu'une composante des techniques de spécification. En offrant des méthodes et des outils de spécification, ces techniques tentent de combler le manque de méthodologie offerte par les langages de spécification quant à la manière de construire les spécifications. En effet, il apparaît que beaucoup de travaux ont porté sur la conception de modèles et de langages de spécification mais très peu d'entre eux apportent des méthodes de construction de spécifications [Dubois 85], [Gruia 85].
De ces travaux, il ressort qu'une "bonne" technique de spécification propose :

- des modèles de spécification permettant la communication entre le spécifieur et le demandeur, ou entre les spécifieurs eux-mêmes,

- un langage de spécification permettant l'expression des modèles du système à développer,

- une méthode de spécification (ou méta-algorithme) permettant de guider le spécifieur par l'application de règles systématiques,

- des outils permettant de vérifier la complétude, la testabilité d'une spécification.

Le projet SACSO (**S**ystème d'**A**ide à la **C**onstruction de **S**pécificati**O**ns) tend à combler les lacunes mentionnées ci-dessus [Dubois 86], [Souquières 86]. Plus précisément, SACSO s'articule autour des éléments suivants :

- l'utilisation d'un langage formel basé sur les types abstraits algébriques, permet de structurer les spécifications tout en autorisant des vérifications telles que la complétude et la cohérence,

- l'existence d'un guide favorisant une obtention systématique de la spécification. Le système propose un ensemble de stratégies (de réutilisation, déductives et inductives) et de méthodes (analyse descendante),

- la présence d'outils tels un pilote d'aide à la construction de spécifications paramétré par des méthodes, ou un outil de production de maquettes (par exécution symbolique, ou par prototypage).

Parallèlement, de nombreux travaux sont effectués actuellement pour améliorer le domaine complexe de la communication homme-machine. En effet, de la qualité de cette communication dépend le succès du système informatique. La conception d'interfaces interactives est dans sa phase initiale de développement avec pour conséquences l'absence d'outils pour l'écriture de telles interfaces et l'existence de principes encore mal établis [Coutaz 85]. Cependant, parmi ces principes, le graphisme apparaît comme un moyen efficace pour améliorer l'interaction homme-machine. Avec l'apparition de matériel permettant d'exploiter pleinement l'utilisation de graphiques, cette tendance n'a fait que se renforcer.

Le mérite premier de l'interaction graphique est d'apparaître plus naturelle à l'utilisateur. D'autre part, l'entrée d'information sous forme graphique dans certains systèmes tels par exemple les systèmes d'aide à la conception de diagrammes de flux, apparaît plus appropriée que l'entrée d'information sous forme textuelle. C'est ainsi que des éditeurs graphiques dirigés par la syntaxe se font de plus en plus nombreux. En sortie, l'utilisation de graphiques permet à l'utilisateur d'avoir une perception plus claire et globale de l'information.

Dans cet ordre d'idées, le portage du système SACSO sur une station de travail a mis en évidence la nécessité d'une présentation graphique de l'information relative à une spécification. L'utilisateur pourrait alors disposer de plusieurs formalismes concrets pour la présentation d'une même spécification SACSO.

Un premier objetif de ce mémoire a consisté à concevoir et réaliser une modification du multi-fenêtrage existant du système SACSO en vue de profiter au maximum des possibilités d'interaction offertes par les stations de travail (mécanisme de sélection par souris, gestion des fenêtres, etc.).

Un deuxième objectif de ce travail, est de proposer une architecture détaillée d'un outil permettant de visualiser graphiquement des objets et des relations entre ces objets représentés de manière interne au système sous forme d'arbre de syntaxe abstraite [Klint 83].

Pour mener à bien ces deux objectifs, le mémoire est décomposé en trois parties, les deux dernières parties constituant la partie originale du travail.

La première partie est consacrée à une présentation du contexte et de l'environnement de travail. Nous débuterons cette partie par la présentation du système SACSO axée sur les particularités de celui-ci. Un deuxième chapitre résumera les principaux aspects du système X Windows sur lequel notre implémentation du multi-fenêtrage a été basée. Ce système conçu spécifiquement pour profiter de toutes les possibilités d'interaction des stations de travail tend à être reconnu comme un standard en matière d'outils de construction d'interfaces. Les principales caractéristiques que devraient respecter un système comme X Windows seront exposées avant d'en présenter ses particularités et concepts. Nous terminerons cette première partie, par un troisième chapitre consacré au langage de programmation CEYX. Cette sur-couche du langage LE_LISP permet une programmation orientée-objet et est utilisée dans le

système SACSO par toutes les primitives de manipulation d'arbres (arbre de syntaxe abstraite, structure intermédiaire). CEYX ne sera pas présenté uniquement sous son aspect de "super-Lisp"; nous montrerons aussi qu'il offre tout un environnement de programmation indépendant du langage.
Au terme de cette première partie, tous les concepts nécessaires à la compréhension des deux parties suivantes auront été introduits.

La deuxième partie sera consacrée à la présentation d'une boîte à outils pour la construction d'interfaces usager que nous avons réalisée au-dessus du système X Windows, profitant ainsi de son statut de standard de fait tout en offrant des primitives de moins bas niveau. Le chapitre 4 présentera cette boîte à outils et son utilisation pour la modification du multi-fenêtrage de SACSO. Pour réaliser une telle modification, nous avons suivi les étapes classiques du cycle de vie d'un logiciel. En conclusion de ce chapitre, nous exposons les problèmes rencontrés, dus au manque de méthodologies pour la construction d'interfaces et au fait que nous nous situions dans une phase de maintenance.

La troisième partie est consacrée à la présentation d'un outil générique de visualisation graphique d'objets formels.
Un exposé des besoins graphiques de SACSO est présenté au chapitre 5. Pour l'essentiel il s'agit de réaliser un outil permettant la visualisation de l'arbre des types d'un type apparaissant dans une spécification SACSO. Une typologie des problèmes à résoudre pour réaliser un tel outil sera présentée dans ce chapitre.
La solution proposée aux différents problèmes exposés au chapitre 5 est détaillée au chapitre 6. Cette solution ne se limite pas uniquement à la conception d'un outil permettant la visualisation de l'arbre des types d'un type.
Elle est plus générale dans la mesure où nous proposons un outil générique qui peut être instancié à un langage d'interface graphique (L.I.G.). Ce dernier est constitué d'un ensemble de définitions de représentations d'un ensemble de concepts sémantiques. Une représentation de concept sémantique est constituée du triplet

- le concept sémantique à représenter graphiquement,
- la représentation graphique souhaitée pout toute occurrence de ce concept sématique,
- le mode de composition spatiale souhaité pour représenter les occurrences de ce concept sémantique.

Nous proposons un méta-langage permettant de définir un L.I.G. Un L.I.G. peut alors être utilisé pour décrire la représentation graphique d'objets et de relations modélisées dans un arbre abstrait. En définissant un L.I.G. adéquat, il est possible de définir la représentation graphique souhaitée pour l'arbre des types d'un type d'une spécification SACSO.
D'autres propositions de solutions plus générales, telles l'annotation graphique d'une structure intermédiaire graphique combinée à un mécanisme d'holophraste permettant l'affichage d'une représentation graphique dans un espace de taille réduite, sont également exposées.

PARTIE I

CONTEXTE ET ENVIRONNEMENT DE TRAVAIL

Chapitre 1

SACSO : un Système d'Aide à la Conception de SpécificatiOns

1.1 Introduction

Ce chapitre présente le système SACSO [Dubois 86], [Dubois 87a], [Lévy 87], [Souquières 86] développé au **C**entre de **R**echerches en **I**nformatique de **N**ancy (CRIN) où nous avons effectué un stage de cinq mois. La section 1.2 présente les objectifs du projet SACSO, la section 1.3, le langage de spécification SACSO, la section 1.4, les méthodes qui peuvent être intégrées à SACSO, la section 1.5, les outils de SACSO et nous terminerons par quelques détails de réalisation en section 1.6, l'interface de SACSO en section 1.7 et un exemple de spécification réalisée sur SACSO en section 1.8.

1.2 Les objectifs

Le projet SACSO s'intéresse à la construction, présentation et validation de spécifications fonctionnelles formelles (cfr figure 1.2.1).

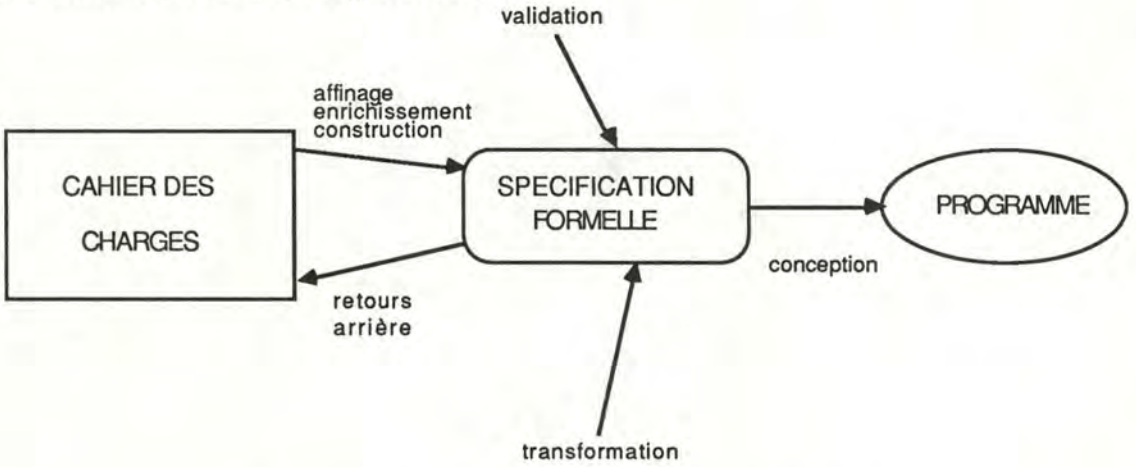


Figure 1.2.1 : construction d'une spécification

Les opérations d'enrichissement complètent la spécification à l'aide d'informations prises dans le cahier des charges.

Les opérations d'affinage consistent à spécifier certains détails qui avaient été laissés de côté lors des étapes précédentes

Les retours en arrière permettent de revoir le cahier des charges quand des inconsistances ont été détectées ou pour défaire certaines décisions prises lors d'opérations d'enrichissement.

Les opérations de transformation de la spécification permettent :

- d'éliminer des redondances,
- d'augmenter la modularité de la spécification de manière à favoriser son évolutivité,

- de simplifier une spécification pour qu'elle soit plus compréhensible,
- d'augmenter le caractère approprié de certaines structures de données de certaines de ses opérations.

SACSO s'articule autour des éléments suivants :

- un langage formel basé sur les **types abstraits algébriques** permettant une structuration des spécifications tout en autorisant des vérifications telles que la complétude et la cohérence,
- des **concepts de base** pré-spécifiés une fois pour toutes, facilement utilisables par le demandeur et pouvant être composés pour obtenir une spécification,
- un ensemble de **méthodes et de stratégies de construction** utilisables par l'utilisateur pour construire sa spécification,
- un ensemble d'**outils automatiques** mettant l'accent sur la saisie assistée, la présentation et la validation de la spécification.

1.3 Le langage de spécification SACSO

Une spécification constitue le dossier commun à toutes les personnes concernées par la construction, l'utilisation et la maintenance d'un système informatique :

- le demandeur doit comprendre précisément les propriétés et le comportement du système spécifié, sur lesquels il doit mettre son accord,
- le programmeur doit, à partir des propriétés exprimées, choisir la meilleure représentation et disposer de tous les renseignements nécessaires à la réalisation du système,
- par la suite, la spécification est le document de base des dossiers de programmation indispensables à la maintenance.

Destinée à être lue par plusieurs catégories de personnes, une spécification SACSO :

- comporte une introduction contenant une description des opérations et des types les plus importants intervenant dans la spécification et permet d'en avoir une vision globale,
- comporte une partie langage naturel, appelée lexique, permettant de comprendre de manière intuitive la structure des objets manipulés ainsi que l'action des opérations sur ces objets,
- définit formellement les objets et opérations introduits. En particulier, il est possible de faire des preuves sur ces définitions (preuves de consistance, de complétude, ou vérifications de propriétés). La définition des opérations se fait soit par composition d'opérations plus élémentaires, soit de manière équationnelle,
- est modulaire et hiérarchique.

Une spécification SACSO est composée :

- d'un ensemble de spécifications d'opérations (appelé énoncé du problème),
- d'un ensemble de spécifications de types (appelé univers du problème), un type étant défini par une structure, un invariant et une liste d'opérations associées.

La structure d'un type T est définie à l'aide d'un autre type. Ce dernier est soit un type de base prédéfini, soit un type construit à l'aide d'un constructeur de types, soit un type spécifié ailleurs. Le type T défini partage alors les opérations et l'invariant du type définissant. Il peut être enrichi de nouvelles opérations et comporter un invariant plus restrictif. L'invariant d'un type est un prédicat opérant une restriction sur l'ensemble des objets du type.
Les types de base de SACSO sont :

- ENTIER,
- CAR (caractère),
- CHAINE (chaîne de caractères),
- BOOL (booléen).

Les types paramétrés ou constructeurs de types sont :

- le produit cartésien : $PC[t_1:T_1, t_2:T_2, \ldots t_n:T_n]$
  où t_i désigne le nom du $i^{ème}$ champ du produit cartésien
  et T_i désigne le type de l'élément du $i^{ème}$ champ de nom t_i du produit cartésien,

- l'ensemble : E[T]
  où T désigne le type des éléments de l'ensemble,

- la suite : S[T]
  où T désigne le type des éléments de la suite,

- la table qui à un objet de type IND (indice) associe un objet de type VAL : T[IND, VAL],

- l'union disjointe : $U[T_1, T_2, \ldots T_n]$
  pour tout type T_i il existe un prédicat qui permet de déterminer si ce type appartient à l'union $U[T_1, T_2, \ldots T_n]$.

Ces constructeurs de types sont munis d'un ensemble d'opérations correspondant aux opérations usuelles y associées. Ainsi, pour le type S (suite), il existe des opérations d'ajout d'un élément en tête de la suite (opération AJOUT), d'accès à un élément de la suite sur la base de son rang (opération ACC), etc ... Pour plus de détails on pourra consulter l'annexe 3.

1.4 Les méthodes

Il existe actuellement peu de méthodes rigoureuses et structurées favorisant la construction correcte de la spécification d'un système. Une méthode est une description précise du processus à suivre pour construire une spécification : l'ordre des actions à effectuer et les types d'objet manipulés concernés. Décrire une méthode particulière consiste à ordonner l'ensemble des actions. Un objectif de SACSO est de proposer des méthodes favorisant la construction et la structuration de la spécification des opérations et des types. L'application de méthodes permet de décrire la suite des décisions prises par le spécifieur pour obtenir la spécification finale par combinaisons d'outils de base prédéfinis.

Chaque méthode peut être vue comme une composition de stratégies. L'application d'une stratégie permet d'expliciter une opération et les types y associés par application d'un certain nombre de règles. Une stratégie est donc un ensemble de règles pour expliciter les opérations et les types. Plusieurs types de stratégies sont envisageables. Par exemple, une stratégie de réutilisation est appliquée à une opération (type) non prédéfini dans les situations suivantes :

(1) l'opération (type) peut être exprimée en terme d'une autre opération (type) spécifiée ailleurs,
(2) l'opération (type) possède une définition implicite dans l'esprit de l'utilisateur final,
(3) l'opération (type) correspond à une opération (type) appartenant à l'environnement extérieur.

Dans le cas (1), l'opération (type) réutilisée correspond soit à une opération (type) prédéfinie, soit une opération (type) complètement spécifiée.

Dans les cas (2) et (3), le spécifieur doit donner une explicitation pour l'opération (type) en terme de combinaisons d'opérations (types) prédéfinies. Les stratégies de réutilisation sont des stratégies terminales dans le processus de spécification ; leur application termine la spécification de l'opération (type) considérée.

De manière analogue, plusieurs méthodes sont envisageables. SACSO intègre actuellement une méthode reposant sur une analyse descendante guidée par les opérations et conduisant à une description en parallèle de l'énoncé et de l'univers.

L'avantage de SACSO est qu'il peut être paramétré par une méthode. Il n'est donc pas limité à une approche spécifique. Pour plus de détails concernant les méthodes nous renvoyons le lecteur à [Dubois 82], [Dubois 84], [Dubois 85], [Dubois 87b].

1.5 Les outils

SACSO a été conçu pour permettre de construire une spécification de manière non linéaire avec possibilité à tout instant d'introduire des définitions incomplètes. Afin de permettre à l'utilisateur de rentrer le minimum d'information, le système, en fonction du contexte propose des déductions. Le droit à l'anomalie sémantique est autorisé : l'anomalie est signalée à l'utilisateur qui n'est pas obligé de la corriger immédiatement. Le système garde la liste des anomalies qui peut à tout moment être consultée par l'utilisateur.

Le système SACSO est un environnement intégré d'outils d'aide à la conception. Ces outils exploitent une base de données commune de spécifications. Les outils peuvent être classés en **outils passifs** (saisie, maintenance, présentation de spécifications ...) et **outils actifs** (pilote, interprète ...). Tous les outils sont conçus pour faciliter la tâche du spécifieur :

- le pilote permet au spécifieur de se définir des guides dans les différentes étapes de construction. A l'aide d'un langage de définition de stratégies, le spécifieur donne l'ordre de construction et précise le niveau d'aide qui doit lui être apporté par le système, ainsi que les traitements qu'il désire effectuer en cas d'erreur. Certaines stratégies sont fournies par le système,
- l'interprète permet de suivre une trace de fonctionnement du système par évaluation symbolique de la spécification,

- les outils de visualisation de la spécification utilisent le multi-fenêtrage pour produire la forme externe de la spécification à l'écran,
- la base de spécifications est hiérarchisée en trois niveaux :

    1. la bibliothèque SACSO, composée des spécifications des types de base, constructeurs de types et opérations y associées,
    2. les spécifications de base du projet qui peuvent être considérées comme spécifications de base par rapport à un domaine d'application donné,
    3. les spécifications effectives de l'utilisateur.

Les spécifications de niveau 1 et 2 sont terminées, en ce sens que

- les spécifications qu'elles utilisent sont terminées,
- elles sont correctes d'un point de vue tant syntaxique que sémantique,
- elles sont complètes (tout est défini).

Les spécifications de niveau 3 sont soit terminées soit non terminées, en ce sens que

- les spécifications qu'elles utilisent ne sont pas forcément terminées,
- elles sont syntaxiquement correctes mais pas nécessairement sémantiquement,
- elles peuvent être incomplètes.

1.6 Implémentation

1.6.1 Structuration du système

Le système est constitué de trois couches hiérarchiques. Pour chaque couche les détails de réalisation de la couche inférieure sont cachés.

Couche 3 : SACSO

Cette couche est constituée d'un ensemble d'outils indépendants non hiérarchisés :

- le pilote de la construction de la spécification guidant l'utilisateur lors de la construction de la spécification,
- les outils de décompilation textuelle permettant de visualiser une spécification de la base de données des spécifications,
- l'interprète permettant de faire de l'évaluation symbolique de spécifications,
- l'outil de production de rapports.

Couche 2 : création, modification, vérification, interrogation et affichage

Ce sont les fonctions de manipulation associées à la représentation interne sous forme d'arbre de syntaxe abstraite. Il y a trois types de fonctions :

- fonctions de création et modification de noeuds d'un tel arbre,

- fonctions d'interrogation : gestion des requêtes élémentaires au niveau de la base des spécifications,
- fonctions de vérification : vérifications locales et globales de spécifications.

Couche 1 : primitives de base de gestion de la représentation interne

Ce sont les fonctions de manipulation associées aux feuilles de la représentation interne sous forme d'arbre de syntaxe abstraite. Il y a trois types de fonctions :

- fonctions de création et modification des diverses feuilles de l'arbre,
- fonctions de manipulation et de recherche permettant de se déplacer dans la représentation interne,
- fonctions de contrôle de base qui assurent des contrôles d'intégrité, de consistance et de complétude au niveau de la base de spécifications.

Couche 0 : Base de données de spécifications

1.6.2 Réalisations

Initialement, SACSO a été installé sur SM90 (système SMX). Le code est écrit en CEYX-LE_LISP (cfr chapitre 3). CEYX est utilisé pour gérer la représentation interne des spécifications sous forme d'arbres. Par la suite, SACSO a été porté sur une station SUN, système UNIX [Kernighan 86], sans modification majeure.

1.7 Le gestionnaire de multi-fenêtrage et les commandes SACSO

1.7.1 Introduction

L'objectif de cette section n'est pas de décrire en détail les commandes, et touches de contrôle nécessaires à l'utilisation de SACSO, mais de présenter plus en détail l'interface usager de SACSO avant de présenter la manière dont nous l'avons modifié (cfr chapitre 4).

Pour plus de détails concernant les commandes, on consultera le manuel d'utilisation de SACSO.

1.7.2 Présentation de l'interface usager de SACSO

Le gestionnaire de multi-fenêtrage ainsi que les modules de gestion des entrées et des sorties de SACSO sont écrits en LE_LISP. Le gestionnaire de multi-fenêtrage offre à l'utilisateur la possibilité de détruire des fenêtres (toutes les fenêtres ne peuvent pas être détruites ; un exemple de fenêtre protégée est la fenêtre "menu"). On peut également déplacer des fenêtres, les placer au sommet de la pile de fenêtres, les placer au bas de cette pile.

Une session-type sur le système SACSO se déroule en deux niveaux. Le premier niveau regroupe les commandes nécessaires à la gestion de la base de données de spécifications, comme par exemple détruire ou créer une spécification. Le deuxième niveau reprend les commandes disponibles permettant de travailler sur une spécification.

Chaque écran de l'interface SACSO a une présentation standardisée avec des zones qui se retrouvent à chaque niveau. La description générale d'un écran est présentée à la figure 1.7.1.

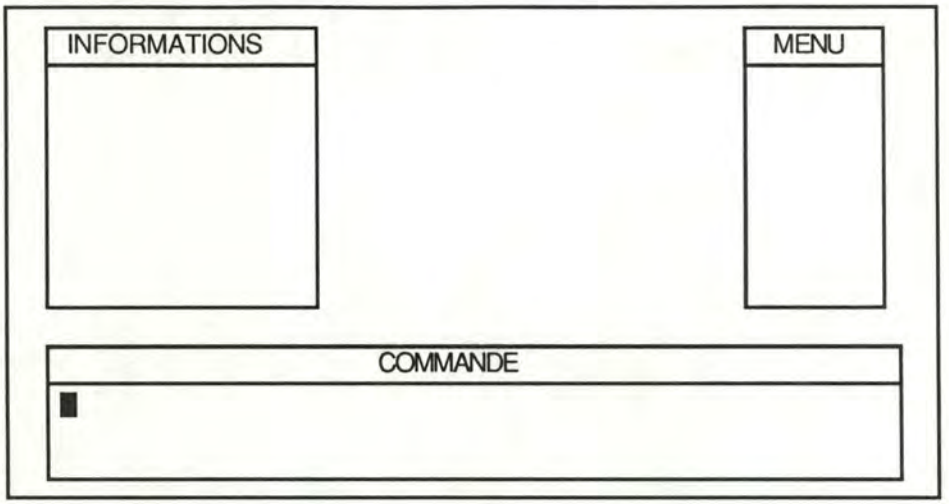


Figure 1.7.1 : présentation d'un écran standardisé de SACSO

La fenêtre MENU contient toutes les commandes activables à l'étape en cours.

La fenêtre COMMANDE permet de lancer des commandes ou de rentrer de l'information. Il s'agit en fait d'un éditeur avec les fonctions classiques d'effacement de caractères, de saut en fin de ligne etc ...

La (les) fenêtre(s) INFORMATIONS contien(nen)t des informations affichées par le système ou à la demande de l'utilisateur. Lorsque le système est lancé, l'écran du premier niveau s'affiche à l'écran (cfr figure 1.7.2).

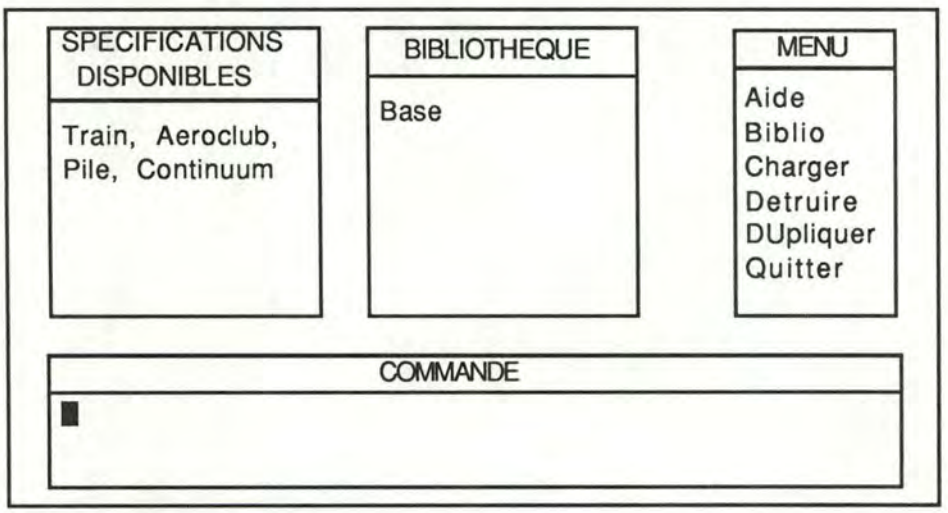


Figure 1.7.2 : écran du niveau 1 de SACSO

La fenêtre BIBLIO contient la liste des noms des spécifications qui sont en bibliothèque.

La fenêtre SPECIFICATIONS DISPONIBLES contient la liste des noms de spécifications qui peuvent être consultées et/ou modifiées.

Les noms des commandes activables sont indiquées dans la fenêtre MENU.
Pour lancer une commande l'utilisateur a deux possibilités :

- soit déplacer le curseur à l'aide de touches de contrôle jusqu'à la fenêtre MENU. Ensuite, déplacer le curseur dans la fenêtre MENU jusqu'au moment où le nom de la commande qu'il veut sélectionner s'affiche en vidéo inverse. Il lui reste alors à lancer cette commande en appuyant sur la touche prévue à cet effet. Si la commande est sans argument elle sera exécutée, sinon l'utilisateur devra introduire le ou les arguments dans la fenêtre COMMANDE,
- soit introduire directement le nom de la commande et le ou les arguments dans la fenêtre COMMANDE grâce à l'éditeur de texte, ou bien uniquement l'abréviation de la commande.

Les commandes disponibles au niveau 1 sont les suivantes :

- **Aide** : cette commande permet d'avoir de l'aide sur un sujet particulier spécifié par l'argument. Le texte d'aide est affiché dans une fenêtre particulière créée à cet effet avec possibilité de faire défiler les pages si le texte est trop long pour la hauteur de la fenêtre,
- **Biblio** : cette commande permet d'ajouter la (les) spécifications donnée(s) en argument(s) en bibliothèque,
- **Charger** : cette commande a deux effets possibles :

    + si l'argument est un nom de spécification existante, cette spécification est chargée en mémoire et peut être consultée et/ou modifiée,
    + sinon la spécification de nom désignée par le paramètre est créée et peut être modifiée.

- **Détruire** : cette commande permet de détruire les spécifications données en argument,
- **Dupliquer** : cette commande permet de dupliquer les spécifications données en argument,
- **Quitter** : cette commande permet de quitter SACSO.

Toutes les erreurs qui peuvent être commises, quel que soit le niveau et la nature de ces erreurs, sont affichées dans une fenêtre spéciale ERREURS qui est créée pour la circonstance.

Lorsque l'utilisateur a lancé avec succès la commande Charger il passe au deuxième niveau. L'écran du niveau 2 est présenté à la figure 1.7.3.

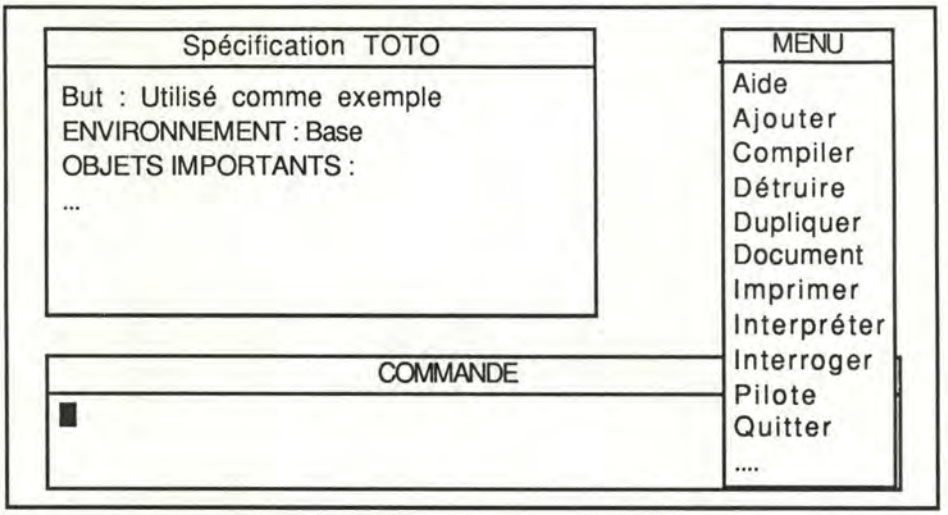


Figure 1.7.3 : écran du niveau 2 de SACSO

La fenêtre SPECIFICATION contient les informations sur la spécification chargée.

La fenêtre MENU contient les commandes disponibles à ce niveau. Ces commandes sont les suivantes :

- **Aide** : cette commande est identique à la commande Aide du premier niveau,
- **Ajouter** : cette commande permet d'ajouter un morceau de spécification,
- **Compiler** : cette commande permet de traduire la spécification en LE_LISP et range la traduction dans un fichier,
- **Détruire** : cette commande permet de détruire un morceau de spécification,
- **Dupliquer** : cette commande permet de dupliquer une opération ou un type,
- **Document** : cette commande permet d'obtenir une sortie sur papier de l'état d'avancement de la spécification, des incomplétudes, des erreurs,
- **Imprimer** : cette commande permet d'imprimer une partie de la spécification courante sur papier,
- **Interpréter** : cette commande permet d'interpréter des morceaux de spécification,
- **Interroger** : cette commande permet de demander des renseignements sur la spécification courante,
- **Pilote** : cette commande permet de construire une spécification tout en étant guidé par une stratégie choisie,
- **Quitter** : cette commande permet de quitter le deuxième niveau, pour se retrouver au premier niveau,
- **Renommer** : cette commande permet de renommer toutes les occurrences d'une opération ou d'un type dans la spécification ou toutes les occurrences d'un objet dans un bloc,
- **Sauver** : cette commande permet de sauvegarder la spécification,
- **Système** : cette commande permet d'activer ou désactiver les contrôles sémantiques et/ou l'aide.

Au niveau de la gestion des fenêtres et de la sélection par curseur, les possibilités sont les suivantes. L'utilisateur peut déplacer le curseur dans n'importe quelle fenêtre et se positionner sur une entité (morceau de texte) qui se trouve dans la fenêtre. Cette entité s'affiche alors en

vidéo inverse. Suivant que la fenêtre est une fenêtre MENU ou une fenêtre INFORMATIONS, l'utilisateur peut effectuer des opérations différentes. Dans le cas d'une fenêtre MENU, l'utilisateur peut déplacer le curseur sur les différentes sélections du menu. Lorsque le curseur est positionné sur une sélection, celle-ci s'affiche en vidéo inverse. Dès que l'entité désirée est sélectionnée, l'utilisateur peut lancer l'exécution de la commande qui correspond à cette entité.

Dans le cas d'une fenêtre INFORMATIONS, l'utilisateur peut aussi se déplacer dans la fenêtre en sélectionnant grâce à des touches de contrôle, les différentes entités de la fenêtre. Il peut alors obtenir plus ou moins de détail sur tout ce qui est affiché dans la fenêtre en appuyant sur une touche de contrôle, obtenir plus de détail sur l'entité affichée en vidéo inverse ou modifier cette entité grâce à l'éditeur de texte. L'information plus ou moins détaillée est alors affichée dans la même fenêtre dans le premier cas et dans une autre fenêtre dans le deuxième cas.

Pour la gestion des fenêtres, l'utilisateur peut en principe détruire toutes les fenêtres. Seules les fenêtres COMMANDE et MENU ne peuvent être détruites, quel que soit le niveau. De plus, au niveau 1, les fenêtres Spécifications disponibles et Biblio ainsi que la fenêtre Spécification au niveau 2 ne peuvent être détruites.

Les fenêtres d'erreurs qui sont créées par le système en cas d'erreur de l'utilisateur sont détruites dès que l'utilisateur appuie sur une touche.

1.8 Exemple de spécification : gestion d'un continuum

L'objectif est de définir un système automatique permettant la gestion d'un forum de discussions (continuum) entre utilisateurs [Dubois 86]. Chaque utilisateur peut rajouter ou lire une transaction. Lors de la lecture, c'est la plus vieille transaction non encore lue par l'utilisateur, si elle existe, qui sera affichée.

Remarques : Lors de la première lecture d'un utilisateur, la première transaction est affichée. Par la suite, l'utilisateur a une position propre dans la file des transactions du continuum.

Type des objets manipulés :

TYPE **CONTINUUM** : composé d'un forum et d'une table d'utilisateurs,
TYPE **FORUM** : un forum est une suite de transactions,
TYPE **TRANSACTION** : texte d'une transaction,
TYPE **UTILISATEURS** : utilisateurs auxquels sont associés leur dernier message lu,
TYPE **NOM-UTIL** : nom d'un utilisateur.

Profil des opérations introduites :

CONT-VIDE : -> **CONTINUUM**
ENVOYER : **CONTINUUM, TRANSACTION** -> **CONTINUUM**
LIRE : **CONTINUUM, NOM-UTIL** -> PC [**CONTINUUM, TRANSACTION**]

Les opérations des types et constructeurs de types prédéfinis utilisés sont les suivants :

- pour le type de base **ENTIER** :
  + et <

infeg (inférieur ou égal) : **ENTIER, ENTIER -> BOOL**
{déterminer si le premier argument de type ENTIER est inférieur ou égal au deuxième argument de type ENTIER}

- pour le constructeur de types TABLE (T) :

insert : T[**F-ELEM, F-ELEM1**], **F-ELEM, F-ELEM1** -> T[**F-ELEM, F-ELEM1**]
{insertion dans une table, à un indice de type F-ELEM, d'un élément de type F-ELEM1}

mod : T[**F-ELEM, F-ELEM1**], **F-ELEM, F-ELEM1** -> T[**F-ELEM, F-ELEM1**]
{modification dans une table, à un indice de type F-ELEM, d'un élément de type F-ELEM1}

acct : T[**F-ELEM, F-ELEM1**], **F-ELEM -> F-ELEM1**
{accès dans une table à un élément de type F-ELEM1 sur la base de son indice de type F-ELEM}

appt : T[**F-ELEM, F-ELEM1**], **F-ELEM -> BOOL**
{appartenance d'un indice donné de type F-ELEM à une table}

- pour le constructeur de types SUITE (S) :

ajout : S[**F-ELEM**], **F-ELEM** -> S[**F-ELEM**]
{adjonction d'un élément de type F-ELEM en tête de suite}

acc : S[**F-ELEM**], **ENTIER -> F-ELEM**
{accès dans une suite à un élément de type F-ELEM sur la base de son rang de type ENTIER}

taille : S[**F-ELEM**] -> **ENTIER**
{nombre d'éléments d'une suite}

- pour le constructeur de types PRODUIT CARTESIEN (PC) :

L'opération de construction sera denotée par $< el_1, el_2, ..., el_n >$ où el_i est la valeur du $i^{ème}$ champ du produit cartésien.

Soit T = PC[el_1 : $\mathbf{EL_1}$, el_2 : $\mathbf{EL_2}$, ... , el_n : $\mathbf{EL_n}$]
L'expression el_i (T) représente la valeur du champ de nom el_i de type EL_i du produit cartésien.

La spécification du problème de gestion d'un continuum se présente alors comme suit. Nous présentons d'abord les types ensuite les opérations selon une stratégie descendante.

TYPE **CONTINUUM**

Partie lexique

But : un continuum est composé d'un forum et d'une table d'utilisateurs.

Invariant : un utilisateur ne peut avoir lu plus de transactions qu'il n'y en a dans le continuum.

Partie formelle

Opérations associees : CONT-VIDE, ENVOYER, LIRE

Structure : PC [FORUM : **FORUM**, UTIL : **UTILISATEURS**]

Invariant : (acct (UTIL (cont), nom)) infeg (taille (FORUM (cont)))

TYPE **FORUM**

Partie lexique

But : un forum est une suite de transactions.

Partie formelle

Structure : S [**TRANSACTION**]

TYPE **TRANSACTION**

Partie lexique

But : une transaction est constituée d'une suite de caractères.

Partie formelle

Structure : S [**CAR**]

TYPE **NOM-UTIL**

Partie lexique

But : un nom d'utilisateur est constitué d'une suite de caractères.

Partie formelle

Structure : S [**CAR**]

TYPE **UTILISATEURS**

Partie lexique

But : utilisateurs d'un continuum, auxquels est associé le nombre de transactions lues.

Partie formelle

Structure : T [**NOM-UTIL, ENTIER**]

OPERATION CONT-VIDE : -> **CONTINUUM**

Partie lexique

But : création d'un continuum vide.

Objets :

cont : **CONTINUUM**

Partie formelle

Définition :

cont = CONT-VIDE ()
= < tvide, svide >

OPERATION ENVOYER : **CONTINUUM, TRANSACTION -> CONTINUUM**

Partie lexique

But : adjonction d'une transaction à un continuum.

Objets :

cont, cont' : **CONTINUUM**	(continua avant et après envoi de la transaction)
trans : **TRANSACTION**	(transaction à ajouter au continuum)
for : **FORUM**	(forum enrichi de la transaction)
ut : **UTILISATEURS**	(utilisateurs du continuum)

Partie formelle

Définition :

cont' = ENVOYER (cont, trans)
= < for, ut >

avec

for = ajout (FORUM (cont), trans)
ut = UTIL (cont)

OPERATION LIRE :
CONTINUUM, NOM-UTIL -> PC [CONTINUUM, TRANSACTION]

Partie lexique

But : lecture d'une transaction

Objets :

res : PC[**CONTINUUM, TRANSACTION**] (continuum mis à jour et transaction lue)
trans : **TRANSACTION** (transaction lue)
cont, cont' : **CONTINUUM** (continua avant et après lecture de la transaction)
nom : **NOM-UTIL** (nom de l'utilisateur désirant lire une transaction)
for : **FORUM**
util, util' : **UTILISATEURS**
nblus, nblus' : **ENTIER** (nombre de transactions lues par l'utilisateur avant et après la lecture)

Partie formelle

Définition :

res = LIRE (cont, nom)
= < cont', trans >
avec
cont' = < for, util'>
trans = acc (for, nblus')
for = FORUM (cont)
util' = si appt (util, nom)
alors modif (util, nom, nblus')
sinon insert (util, nom, l)
nblus' = si nblus < taille (for)
alors nblus + 1
sinon nblus
util = UTIL (cont)
nblus = si appt (util, nom)
alors acct (util, nom)
sinon 0

Nous présentons dans la suite, un outil de construction d'interfaces que nous avons utilisé pour construire un nouveau gestionnaire de fenêtres pour SACSO.

Chapitre 2

X Window System : un outil de construction d'interfaces homme-machine

2.1 Introduction

Ce chapitre, présente le système X Windows (X), [Gettys 86], [XWindows 87], les concepts associés et les facilités de bas niveau disponibles pour construire des applications interactives. X a acquis une grande popularité, surtout dans la communauté UNIX, et est actuellement considéré comme un standard. X est basé sur un protocole réseau, à savoir, la communication inter-processus par "stream" remplace le traditionnel appel de procédure. Une application peut utiliser des fenêtres sur n'importe quel écran dans un réseau X. X a été conçu dès le départ avec un objectif de portabilité, d'où sa grande popularité.

Nous allons d'abord présenter les caractéristiques que devraient avoir un système de fenêtrage, ensuite nous décrirons le modèle sous-jacent au système X (section 2.3), la hiérarchie de fenêtres (section 2.4), les caractéristiques propres à X par niveau croissant d'abstraction (section 2.5 à 2.12) et quelques éléments d'évaluation (section 2.13).
La version décrite ici est la version 10.

2.2 Caractéristiques souhaitables pour un système de fenêtrage

Un système de fenêtrage contient plusieurs interfaces avec son environnement. Tout d'abord, une **interface de programmation** se présente comme une librairie de routines et de types fournis dans un langage de programmation pour interagir avec le système de fenêtrage. Généralement, des interfaces de bas niveau (dessiner un point, une droite, ...) ainsi que des interfaces de haut niveau (menus, boîtes à messages, ...) sont fournies. Ensuite, **l'interface de l'application** prend en compte les interactions de l'utilisateur avec l'apparence visuelle (interface à l'écran) qui est spécifique à l'application. Enfin, **l'interface de gestion** prend en compte les interactions de l'utilisateur avec l'espace de travail et les organes d'entrée (clavier, souris, ...). Cette interface définit comment les applications sont disposées à l'écran et comment l'utilisateur peut passer d'une application à l'autre. L'interface individuelle de chaque application définit comment l'information est présentée et manipulée à l'intérieur de l'application. **L'interface utilisateur** est la réunion de toutes les interfaces des différentes applications et des interfaces de gestion.

A côté des applications, il y a trois composants majeurs d'un système de fenêtrage :

- le *gestionnaire de fenêtres* : celui-ci implémente la partie gestion de l'espace de travail de l'interface de gestion. Il contrôle entre autres la taille et le placement des fenêtres de l'application,
- le *gestionnaire des entrées* : celui-ci implémente la partie gestion des entrées de l'interface de gestion. Il contrôle le clavier et la souris,

- le *système de fenêtrage de base* : les applications et les gestionnaires de fenêtres sont construits sur ce système de base. Le système de base est principalement chargé de la gestion de la hiérarchie de fenêtres, de la gestion des ressources et de l'établissement des connexions entre clients et serveurs.

Pour devenir un standard, les caractéristiques du système de base devraient être les suivantes :

1. Le système doit être implémentable sur une grande variété d'écrans et doit pouvoir travailler sur des écrans bitmap et des organes d'entrée de types différents.
2. Les applications doivent être indépendantes du matériel. Cette indépendance comprend plusieurs aspects :

   - il faut éviter de devoir recompiler, réécrire ou même relier une application pour chaque nouveau matériel graphique,
   - toutes les fonctions graphiques doivent pouvoir travailler sur chaque écran supporté,
   - les applications doivent pouvoir travailler aussi bien sur des écrans monochromes que sur des écrans couleurs.

3. Le système doit être transparent au niveau du réseau. Une application tournant sur une machine doit être capable d'utiliser un écran d'une autre machine, sans que ces deux machines n'aient besoin d'avoir la même architecture ou le même système d'exploitation.

4. Le système doit tolérer plusieurs applications affichant de manière concurrente. Il doit être possible par exemple, d'afficher un moniteur de performance à l'écran, tout en travaillant sur une fenêtre éditeur de texte.

5. Le système doit pouvoir accueillir de nombreuses applications et interfaces de gestion différentes. Il doit permettre la construction d'une grande variété d'interfaces car chacun a sa propre vue d'une interface. Par exemple, au lieu de fournir une politique de gestion fixe des menus, il faut pouvoir offrir des primitives à partir desquelles chacun peut construire des menus comme il le désire.

6. Le système doit permettre le recouvrement de fenêtres, ainsi que l'affichage dans des fenêtres recouvertes en partie. Ceci résulte du point précédent.

7. Le système doit permettre une hiérarchie de fenêtres redimensionnables et une application doit pouvoir utiliser plusieurs fenêtres à la fois. En fournissant une hiérarchie de fenêtres, on évite à l'application de dupliquer la gestion des entrées pour chaque fenêtre.

8. Le sytème doit être performant, offrir une haute qualité pour les textes, graphiques en deux dimensions et images. L'utilisation de modèles de haut niveau, par lesquels l'application décrit ce qu'elle veut en termes d'objets abstraits tandis que le système détermine comment rendre le mieux possible l'image, ne peut être imposé comme la

seule forme d'interface graphique. En effet, aucun modèle ne reçoit l'unanimité des préférences des utilisateurs. Il est important de fournir de tels modèles mais ils doivent être construits en couches au dessus du système de base. La solution la plus générale est de laisser le soin à l'application de se débrouiller.

9. Le système doit être extensible. Le noyau doit pouvoir être étendu pour ajouter de nouvelles fonctionnalités.

2.3 Le modèle du système X

Le système X Windows est basé sur le **modèle client-serveur**. Ceci découle des exigences 2 et 3 de la section 2.2. Pour chaque écran physique il y a un serveur qui le contrôle. Une application cliente et un serveur communiquent par une liaison duplex.

Si le client et le serveur sont sur la même machine, alors le "stream" est un mécanisme de communication local inter-processus (IPC) ; dans le cas contraire une connexion réseau est établie entre la paire.

Plusieurs clients peuvent avoir une connexion ouverte simultanément avec un serveur, et un client peut avoir plusieurs connexions ouvertes vers plusieurs serveurs simultanément.

La tâche essentielle du serveur est de répercuter les requêtes des clients à l'écran, et de transmettre les commandes effectuées par un utilisateur grâce au clavier et à la souris au client approprié. Un serveur est implémenté comme un processus séquentiel, utilisant le principe d'ordonnancement à tour de rôle (round-robin scheduling) entre clients. Le serveur n'est pas placé dans le noyau du système d'exploitation pour des raisons de facilité de maintenance. Le serveur comprend le système de fenêtrage de base.

Toutes les dépendances de ce dernier sont dans le serveur, tandis que le protocole de communication entre le client et le serveur est indépendant du matériel. Dès lors l'addition d'un nouvel écran requiert simplement l'addition d'un nouveau serveur.

2.3.1 Le protocole de communication

Le protocole de communication entre serveur et client a été conçu de telle manière que si le serveur doit attendre une réponse du client, il doit pouvoir continuer à servir d'autres clients.

2.3.2 Les ressources

Les ressources de base fournies par un serveur sont les fenêtres, les polices de caractères, curseurs, et images. Les clients demandent la création d'une ressource en donnant les paramètres appropriés. Dans le cas d'une fenêtre par exemple ce sera la taille de la fenêtre, le type de la fenêtre, la fenêtre père. Le serveur alloue la ressource et renvoie un identificateur. L'utilisation et l'interprétation de ces identificateurs est indépendante de la connexion réseau. Par conséquent, tout client connaissant l'identificateur de la ressource peut utiliser cette ressource même si il ne l'a pas créée. Pour éviter le problème de clients qui ne détruisent pas leurs ressources, la durée de vie maximum d'une ressource est celle de la connexion au travers de laquelle elle a été créée. Un contrôle d'accès est effectué uniquement lorsqu'un client essaye d'établir une connexion au serveur. Ce contrôle est basé sur l'adresse du réseau hôte. Une fois ce contrôle effectué, le client peut manipuler librement les ressources. Le format d'une requête client est simple. Il s'agit d'un bloc de données avec un certain nombre de paramètres de longueur fixe, et éventuellement des paramètres de longueur variable. Par exemple pour

afficher du texte dans une fenêtre, le bloc de données se présente comme décrit à la figure 2.3.1.

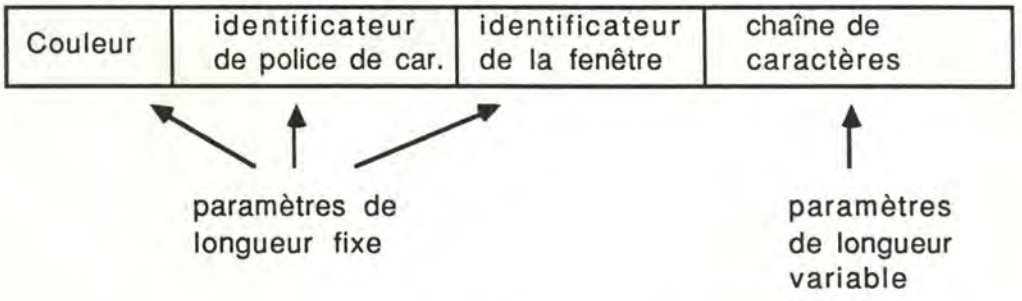


Figure 2.3.1 : bloc de données pour afficher une chaîne de caractères dans une fenêtre

Une caractéristique particulière de X est que contrairement à beaucoup de systèmes de fenêtrage basés sur UNIX il n'utilise pas un descripteur de fichier pour représenter une fenêtre. X ne limite donc pas le nombre de fenêtres qui peuvent être ouvertes simultanément.

2.4 La hiérarchie de fenêtres

Les fenêtres sont organisées en une hiérarchie basée sur une relation de couverture spatiale. Au sommet de cette hiérarchie se trouve la **fenêtre racine** (rootwindow) qui couvre l'écran entier. Toutes les fenêtres qui sont créées sont des sous-fenêtres de la fenêtre racine. La hiérarchie de fenêtres modélise les piles de papier du bureau. Pour une fenêtre donnée, ses sous-fenêtres peuvent être empilées dans n'importe quel ordre, avec des recouvrements arbitraires. Quand une fenêtre F1 couvre partiellement ou complètement une fenêtre F2, on dit que F1 **obscurcit** F2.

Une fenêtre peut être "visible" (mapped) à l'écran, si tous ses ancêtres sont aussi "visibles" ; elle peut aussi être "invisible" (unmapped). Une sortie vers une fenêtre feuille (sans sous-fenêtres) est toujours coupée à la portion visible de la fenêtre ; un dessin dans une telle fenêtre n'est jamais produit dans une fenêtre qui l'obscurcit.

Une sortie vers une fenêtre non feuille (avec sous-fenêtres) peut être effectuée selon deux modes. En mode coupé (clipped), la sortie est coupée par toutes les fenêtres obscurcissantes, mais en mode "dessiner à travers" (drawthrough) la sortie n'est pas coupée par les sous-fenêtres. Le système de coordonnées est défini avec l'axe des x horizontalement et l'axe des y verticalement. Chaque fenêtre a son propre système de coordonnées ; ceci est nécessaire si l'on veut ne pas s'occuper de la position à laquelle la fenêtre se trouve lorsqu'on affiche dans cette fenêtre.

Une fenêtre peut avoir un bord (border) qui est un cadre qui entoure la fenêtre. Les opérations possibles sur les fenêtres sont les suivantes :

- création d'une fenêtre "invisible" en donnant la fenêtre parent, la position en x et en y du coin supérieur gauche de la nouvelle fenêtre dans la fenêtre parent, la hauteur et la largeur de la fenêtre (primitive XCreateWindow),
- destruction d'une fenêtre ce qui a pour effet de détruire toutes les fenêtres en dessous de cette fenêtre dans la hiérachie (primitive XDestroyWindow),

- rendre "visible" (primitive XMapWindow) ou "invisible" (primitive XUnmapWindow) une fenêtre,
- déplacer une fenêtre (primitive XMoveWindow),
- changer la taille d'une fenêtre en donnant la largeur et la hauteur souhaitées (primitive XChangeWindow),
- placer une fenêtre au sommet de la pile de fenêtres (primitive XRaiseWindow),
- placer une fenêtre au bas de la pile de fenêtres (primitive XLowerWindow).

L'écran d'une station de travail modélise un bureau. La pile de fenêtres est une structure de données utilisée pour modéliser une pile de documents sur le bureau, chaque fenêtre modélisant un document.

X fournit deux types de fenêtres. Les fenêtres **opaques** qui obscurcissent les sorties et la visibilité d'autres fenêtres. Les fenêtres **transparentes** qui sont toujours invisibles à l'écran et qui n'obscurcissent pas les sorties, ou la visibilité d'autres fenêtres. Une sortie vers une fenêtre transparente est coupée à la taille de la fenêtre mais est dessinée sur la fenêtre parent.

2.5 Couleurs

X offre la possiblité d'utiliser des couleurs. Nous ne mentionnerons cette possibilité qu'à titre de complétude.

2.6 Graphiques et texte

Les opérations graphiques dans X sont exprimées en terme de concepts d'assez haut niveau, tels des lignes, rectangles, courbes et polices de caractères.

2.6.1 Les images

Deux types d'image stockés en mémoire existent en X. Un **bitmap** est un rectangle à plan unique. Un **pixmap** est une matrice à N lignes où N est le nombre de bits par pixel utilisés par un écran particulier. Un bitmap ou un pixmap peuvent être créés en transmettant tous les bits au serveur. Un pixmap peut aussi être créé en copiant une région rectangulaire d'une fenêtre.

Les bitmaps peuvent être utilisés comme des masques ou pour construire des curseurs ; les pixmaps sont utilisés pour stocker des images fréquemment dessinées. Ces derniers sont aussi utilisés pour stocker temporairement la partie que cache un menu à déroulement lorsqu'il est affiché à l'écran et réafficher cette partie lorsque le menu est effacé. Dans ce type de manipulation où la vitesse est primordiale ces images stockés en mémoire sont souvent utilisés. Cependant les pixmaps sont principalement utilisés comme "tuiles" pour couvrir une région.

2.6.2 Les graphiques

Chaque requête graphique et de texte comprend une **fonction logique**, et un "sélecteur de bits" (plane select mask) utilisé pour préciser les bits du pixel auxquels il faut appliquer la fonction logique. Toutes les seize fonctions logiques (bit source AND bit destination, bit source OR bit destination, ...) sont disponibles. Etant donné un pixel source et un pixel destination, la fonction est exécutée bit par bit sur les bits correspondants au pixel, mais seulement sur les bits spécifiés dans le " sélecteur de bits".

X fournit une primitive complexe pour dessiner des lignes (primitive XLine). Elle permet des combinaisons arbitraires de segments droits ou courbés, définissant des formes ouvertes ou fermées. Les lignes peuvent être continues, pointillées, ou bien avec un certain motif (pattern).

2.6.3 Texte

X fournit un support direct pour des polices de caractères sous forme de bitmaps. Une **police de caractères** est au plus constituée de 256 bitmaps. Les clients peuvent charger ces polices de caractères en les désigant par leur nom (exemple : XGetFont (helv12b)). Certaines polices de caractères sont proportionnelles (tous les caractères n'ont pas la même largeur), d'autres pas ; certaines contiennent des caractères pouvant être écrits en gras, en italique, avec des symboles mathématiques, etc ...

Un chargement de police de caractères peut échouer si la place mémoire est insuffisante. Des buffers sont fournis par le serveur pour supporter des opérations de "**couper-coller**" (cut and paste). Les clients peuvent lire et écrire dans ces buffers un string arbitraire de bytes.

2.7 Les "expositions" (exposures)

Lorsqu'une partie d'une fenêtre cachée par une autre devient visible, on dit qu'une "exposition" s'est produite ; la partie de la fenêtre devenue visible doit être restaurée. En X cela incombe au client et non pas au serveur. Quand une région d'une fenêtre devient exposée, le serveur envoie un événement asynchrone au client spécifiant la fenêtre et la région "exposée". L'application peut alors soit redessiner le contenu de la fenêtre entière, soit seulement la partie de la fenêtre à redessiner. Si le serveur devait s'occuper du problème d'exposition il devrait :

- soit retenir une liste de toutes les requêtes de sortie effectuées sur la fenêtre. Lorsqu'une partie de la fenêtre devient exposée, le serveur réexécute soit les requêtes sur la fenêtre entière, soit uniquement sur les régions de la fenêtre affectées par les requêtes,
- soit en sauvant en mémoire des images. Chaque fois qu'une partie de fenêtre est cachée par une autre, le contenu de cette partie de fenêtre est sauvée en mémoire et les requêtes de sortie sont effectuées à la fois sur la fenêtre et sur la mémoire. Lorsque la partie de fenêtre redevient visible, il suffit de réafficher le contenu qui avait été sauvé en mémoire.

Aucune de ces solutions n'est acceptable. La première parce que le serveur ne sait pas savoir quand des requêtes de sortie émises plus tard rendent nulles les précédentes. Il passerait alors son temps à maintenir des listes extrêmement longues. La deuxième solution pose des problèmes de taille mémoire puisque dans certains cas il faut sauver en mémoire le contenu de tout un écran. Une autre raison pour laisser à l'application le soin de redessiner les parties de fenêtres exposées est que la plupart des applications peuvent tirer avantage de leurs propres structures d'information dans cette tâche.

2.8 Les curseurs

Les clients peuvent créer des curseurs (primitive XCreateCursor) à partir des éléments suivants : un bitmap source, une paire de valeurs de pixel avec lesquels il faut afficher le bitmap, un bitmap masque qui définit la forme précise de l'image et enfin une coordonnée à l'intérieur du bitmap source qui définit le centre du curseur. Une fenêtre "**contient**" la souris

(curseur de la souris) quand le centre du curseur est à l'intérieur d'une portion visible de la fenêtre ou d'une de ses sous-fenêtres. La souris "**est**" dans la fenêtre si la fenêtre mais aucune sous-fenêtre ne contient la souris. Il est possible de définir un curseur différent pour chaque fenêtre. Le serveur affiche automatiquement le curseur associé à la fenêtre dans laquelle est la souris. Ceci permet par exemple d'attacher un curseur approprié à une fenêtre destinée à recevoir du texte. Lorsqu'une fenêtre n'a pas de curseur associé, le serveur utilise le principe de la hiérarchie de fenêtres c'est-à-dire qu'il affiche le curseur de l'ancêtre le plus proche de cette fenêtre pour laquelle un curseur est défini.

2.9 La gestion des entrées

Les entrées sont associées aux fenêtres. Les entrées pour une fenêtre donnée sont contrôlées par un seul client qui ne doit pas nécessairement être celui qui a créé la fenêtre. Les événements générés par le serveur sont classifiés en différents types et le client choisit pour une fenêtre les types d'événements qui l'intéressent (primitive XrInput). Comme exemple de type d'événement, citons le type d'événement "LeaveWindow". Lorsque ce type d'événement est sélectionné pour une fenêtre, le gestionnaire des entrées générera un événement de ce type chaque fois fois que la souris quitte cette fenêtre. Le serveur n'enverra au client que les événements d'un des types sélectionnés. Si un événement d'entrée est généré pour une fenêtre et que le client contrôlant cette fenêtre n'a pas sélectionné ce type d'événement, alors le serveur propage l'événement à la fenêtre ancêtre la plus proche pour laquelle ce type d'événement a été sélectionné. Chaque événement contient l'identificateur de la fenêtre pour laquelle a eu lieu la génération de cet événement.

2.9.1 La souris

Le protocole X est conçu pour des souris ayant jusqu'à trois boutons. Une application peut recevoir des événements relatifs à l'enfoncement ou au relâchement de chaque bouton de la souris. Chaque événement contient les coordonnées locales à la fenêtre et globales à l'écran, l'état courant de tous les boutons ainsi qu'une estampille de temps servant notamment à décider si une succession de clics constitue ou non un double clic. L'application peut aussi recevoir un événement chaque fois que la souris entre (événement de type EnterWindow) ou bien quitte (événement de type LeaveWindow) une fenêtre.

2.9.2 Le clavier

Le client peut recevoir des événements lors de l'enfoncement ou du relâchement d'une touche du clavier. Les événements clavier ne sont pas exprimés en termes de codes ASCII. Chaque touche a un code unique et l'application doit alors traduire ce code pour obtenir le caractère approprié. Le clavier est toujours attaché à une fenêtre appelée la fenêtre "focus". Les événements clavier contiennent aussi l'état des touches Shift, Shiftlock, Control et Meta pour pouvoir prendre en compte des combinaisons de touches. Il y a deux modes de gestion du clavier : "**real-estate**" et "**listener**". En mode "**real-estate**", les entrées clavier sont dirigées vers la fenêtre dans laquelle est la souris, tandis qu'en mode "**listener**" les entrées clavier sont dirigées vers une fenêtre particulière quel que soit l'endroit où se trouve la souris.

2.10 Gestion des fenêtres

Un gestionnaire de fenêtres a comme fonction principale de reconfigurer l'écran c'est-à-dire déplacer certaines fenêtres, les agrandir, les dépiler, les empiler différemment. Il y a deux

grandes classes de gestionnaires de fenêtres : les **manuels** et les **automatiques**. Un gestionnaire **manuel** est passif et offre à l'utilisateur la possibilité de manipuler les fenêtres sur l'espace de travail. La position initiale des fenêtres à l'écran dépend entièrement de l'application. Un gestionnaire **automatique**, par contre est actif et agit pour la plupart du temps sans interaction humaine. L'écran est découpé en fenêtres de telle sorte qu'aucune fenêtre ne recouvre une autre. En X tous les gestionnaires existants sont manuels. Les gestionnaires de fenêtres sont dirigés par la souris et/ou le clavier. Lorsque certains boutons de la souris et/ou certaines touches du clavier sont enfoncés, le serveur passe la main au gestionnaire de fenêtres qui recevra tous les événements jusqu'à ce que le bouton et/ou la touche soient relâchés. Le gestionnaire peut entre-temps réaliser l'action associée à la touche et/ou au bouton. Par exemple, lorsque le curseur est dans la fenêtre, en appuyant sur la touche shift et sur le bouton droit de la souris il est possible de déplacer la fenêtre. En fait, la fenêtre va suivre les déplacements du curseur jusqu'à ce qu'une ou les deux touches soient relâchées.

En réalité, les gestionnaires de fenêtres disponibles en X (uwm, xwm ...) sont construits à partir des primitives de base fournies par le système de fenêtrage de base de X. Il est donc possible de construire soi-même son propre gestionnaire de fenêtres. La **gestion asynchrone** des événements peut causer quelques problèmes. Par exemple, considérons trois fenêtres empilées les unes sur les autres. Supposons qu'un clic de souris sur le bouton droit lorsque la souris est dans la fenêtre consiste à placer la fenêtre au bas de la pile de fenêtres. Si l'utilisateur clique deux fois rapidement lorsque la souris est dans la première fenêtre de la pile de fenêtres, l'effet voulu est de placer au bas de la pile de fenêtres la première fenêtre et la deuxième fenêtre à partir du sommet de la pile. Malheureusement, à cause de la gestion asynchrone des événements ces deux clics vont être interprétés par le gestionnaire de fenêtres comme placer deux fois la première fenêtre au bas de la pile de fenêtres. Lors de la construction d'un gestionnaire de fenêtres il convient donc de se méfier de la gestion asynchrone des événements.

2.11 Gestion des événements par une application

Une application doit gérer les événements qu'elle reçoit de la manière suivante. Les événements sont placés par le serveur dans une **file d'attente** associée à l'application. L'application se contente de lire les événements de la file d'attente et de répondre à chaque événement par l'action correspondante qui doit être réalisée en cas de survenance d'un événement de ce type. Concrètement cette réponse aux événements se fait par une boucle infinie (cycle) où à chaque fois l'application lit un événement dans la file et effectue le traitement correspondant en fonction du type d'événement.

2.12 Principaux concepts graphiques de X Windows : menu, fenêtre, editeurs, boîte à messages

2.12.1 Les fenêtres

Il existe en X Windows plusieurs types de fenêtres :

- fenêtres ordinaires : ces fenêtres peuvent contenir aussi bien du texte que des graphiques ; toutefois, il n'y a pas de buffer associés à ces fenêtres. Une gestion de buffer doit être prévue pour redessiner le contenu de ces fenêtres en cas d'exposition

(cfr section 2.7). Les fenêtres ordinaires seront simplement nommées par la suite "**fenêtres**". Les primitives typiques offertes pour les fenêtres ont déjà été exposées brièvement dans la section 2.10,

- fenêtres prévues pour contenir du texte : ces fenêtres sont conçues spécialement pour pouvoir y afficher du texte. Un buffer contenant le texte affiché dans la fenêtre leur est associé. Ces fenêtres seront nommées par la suite "**fenêtres-texte**". Les primitives typiques pour les fenêtres-texte sont :

  - créer une fenêtre-texte en donnant la hauteur et la largeur souhaitée pour cette fenêtre, la position en x et en y à laquelle elle doit être affichée à l'écran, l'instance de la fenêtre parent dans laquelle elle doit être créée, la police de caractères à utiliser pour le texte qui sera affiché dans la fenêtre, la largeur souhaitée pour le bord de la fenêtre (primitive TextCreate),
  - réafficher le contenu d'une fenêtre-texte en donnant son instance (primitive TextRedisplay),
  - effacer le contenu d'une fenêtre-texte en donnant son instance (primitive TextClear).

2.12.2 Les éditeurs

X Windows distingue un certain nombre d'éditeurs constitués de zones "pointables" ou de boutons dans lesquels l'utilisateur peut aller cliquer ; l'action attachée à la zone "pointable" ou au bouton sera exécutée. Le choix du terme éditeur est quelque peu "malheureux" car sans rapport avec l'acception classique généralement admise pour ce dernier. Les éditeurs de X Windows sont les suivants :

- **pushbutton** : il s'agit d'un éditeur constitué de plusieurs boutons de forme ovale (cfr figure 2.12.1). Dans chaque bouton se trouve une chaîne de caractères représentant un "descriptif" permettant de décrire l'action qui sera effectuée si l'utilisateur sélectionne un de ces boutons. Le bouton dans lequel l'utilisateur clique se colorie en noir lors de l'enfoncement de la touche de la souris. Il reprend sa couleur originelle lors du relâchement de la touche de la souris. Ce type d'éditeur sera appelé par la suite editeur à boutons. Les primitives typiques pour l'éditeur à boutons sont les suivantes :

  - créer un éditeur à boutons en remplissant une structure de données (record) contenant le nombre de boutons souhaités, le nombre de colonnes de boutons souhaitées, les chaînes de caractères constituant les descriptifs des boutons et l'instance de la fenêtre dans laquelle il faut créer l'éditeur à boutons (primitive XrPushButton),
  - rendre un bouton de l'éditeur à boutons "insélectionnable" par l'utilisateur en donnant l'instance et le numéro du bouton de l'éditeur à boutons à rendre "insélectionnable" (primitive XrPushButton).

Figure 2.12.1 : éditeur à boutons (pushbutton)

- **radiobutton** : il s'agit d'un éditeur constitué lui aussi de plusieurs boutons mais de forme ronde (cfr figure 2.12.2). A côté de chaque bouton figure une chaîne de caractères représentant un "descriptif" permettant de décrire l'action qui sera effectuée si l'utilisateur clique sur le bouton correspondant. Le bouton dans lequel l'utilisateur clique se colorie en noir lors de l'enfoncement de la touche de la souris. Mais contrairement à l'éditeur à boutons, le bouton conserve cette couleur lors du relâchement de la touche. De plus, le bouton précédent dans lequel avait cliqué l'utilisateur reprend sa couleur originelle. L'éditeur radio est utilisé dans les cas où il est nécessaire de remémorer à l'utilisateur le dernier bouton dans lequel il a cliqué. Ce type d'éditeur sera appelé par la suite éditeur radio. Les primitives typiques pour l'éditeur radio sont les suivantes :

  - créer un éditeur radio en remplissant une structure de données (record) contenant le nombre de boutons souhaités, le nombre de colonnes de boutons souhaitées, les chaînes de caractères constituant les descriptifs des boutons et l'instance de la fenêtre dans laquelle il faut créer l'éditeur radio (primitive XrRadioButton),
  - rendre un bouton de l'éditeur radio "insélectionnable" par l'utilisateur, en donnant l'instance et le numéro du bouton de l'éditeur radio à rendre "insélectionnable" (primitive XrRadioButton).

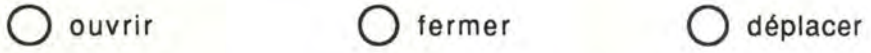


Figure 2.12.2 : éditeur radio (radiobutton)

- **titlebar** : il s'agit d'un éditeur constitué de quatre zones "pointables" auxquelles peuvent être associées une action à exécuter si l'utilisateur clique dans l'une de ces zones (cfr figure 2.12.3). Ces quatre zones sont le titre (en blanc sur fond noir), la bande grise située de part et d'autre du titre et enfin deux boîtes à sélection (gadjetboxes) qui sont facultatives, situées sur le bord gauche et/ou droit du titlebar. Cet éditeur sera appelé par la suite barre de titre. La primitive typique pour la barre de titre est la suivante : créer une barre de titre en donnant un record contenant la chaîne de caractères à afficher dans le titre de la barre de titre, les chaînes de caractères à afficher dans les boîtes à sélection gauche et droite et l'instance de la fenêtre dans laquelle il faut créer la barre de titre (primitive XrTitleBar),

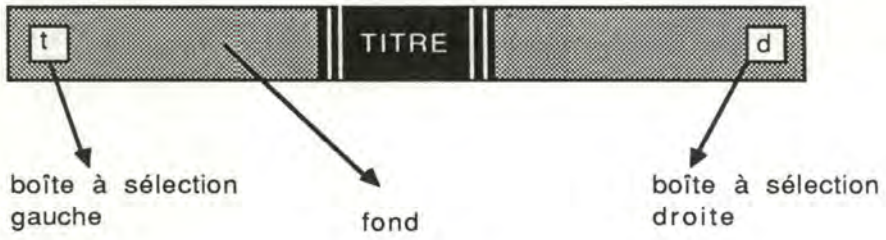


Figure 2.12.3 : barre de titre (titlebar)

2.12.3 L'éditeur de texte

X Windows permet d'utiliser un éditeur de texte classique muni de fonctions standards telles que effacer une ligne, détruire un caractère, ajouter un caractère, positionner le curseur à n'importe quel endroit du texte grâce à la souris etc ... Les primitives typiques pour l'éditeur de texte sont les suivantes :

- créer un éditeur de texte en remplissant une structure de données (record) contenant l'instance de la fenêtre dans laquelle il faut créer l'éditeur de texte, la hauteur et largeur de l'éditeur de texte et enfin la police de caractères à utiliser pour afficher le texte dans l'éditeur (primitive XrPageEdit),
- rendre l'éditeur de texte insensible (sensible) aux actions de l'utilisateur (primitive XrPageEdit), c'est-à-dire inhiber (permettre) les actions de l'utilisateur.

2.12.4 Le menu à déroulement

X Windows permet d'attacher à une fenêtre un menu qui lui même peut être composé de plusieurs sous-menus. Un menu est composé de "descriptifs" ou chaînes de caractères indiquant l'action qui sera éxécutée lors de la sélection de cet item. Un sous-menu est un menu attaché à un item du menu. Les sélections disponibles du sous-menu peuvent être obtenues en positionnant le curseur sur la flèche qui se trouve à côté de l'item auquel est attaché le sous-menu. La primitive typique pour les menus est la primitive de création qui reçoit en paramètre un record contenant le nombre de sélections du menu, les chaînes de caractères représentant les descriptifs des sélections du menu, l'instance de la fenêtre à laquelle il faut attacher le menu (primitive XrMenu).

2.12.5 La boîte à messages

Une boîte à messages (messagebox) est une fenêtre utilisée lorsque l'on veut envoyer un message à l'utilisateur. Elle est composée d'un message qui est une chaîne de caractères représentant le message que l'on veut afficher à l'écran dans la boîte à messages. De plus la possiblité est offerte d'afficher à l'écran dans la boîte à messages un dessin (un pixmap) donnant lui aussi une indication à l'utilisateur. La boîte à messages peut aussi contenir un éditeur à boutons dans le cas où on propose à l'utilisateur une alternative. Ce type de fenêtres sera appelé par la suite boîte à messages (cfr figure 2.12.4). La primitive typique pour les boîtes à messages est la primitive de création qui reçoit en paramètre une structure de données (record) contenant le nombre de boutons souhaités pour l'éditeur à boutons, les chaînes de caractères constituant les descriptifs placés à côté des boutons de l'éditeur à boutons, l'instance du pixmap à placer dans la boîte à messages, l'instance de la fenêtre dans laquelle il faut créer la

boîte à messages et enfin la chaîne de caractères constituant la partie message de la boîte à messages.

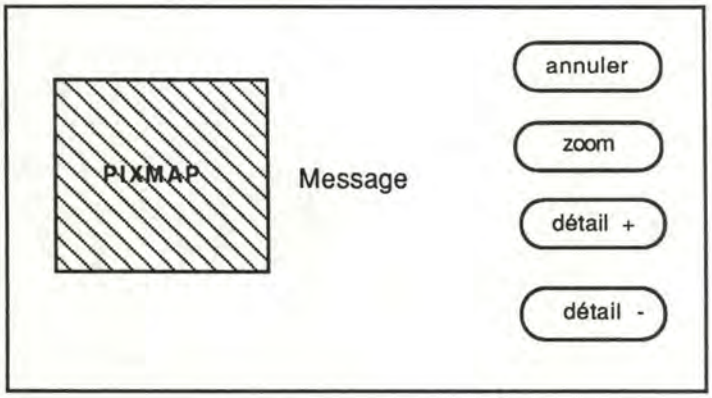


Figure 2.12.4 : boîte à messages (messagebox)

2.13 Evaluation

Cette section reprend ce qui nous paraît être les points faibles et les points forts de X Windows suite à l'utilisation que nous en avons fait.

2.13.1 Les points faibles de X Windows

1) X offre des primitives de trop bas niveau. L'effort de codage que doit fournir le programmeur utilisant X Windows est trop important. Par exemple, un menu ne peut être créé en une seule instruction. Il faut d'abord remplir un record avec les éléments suivants :

   - nombre de "descriptifs" du menu,
   - chaînes de caractères constituant les descriptifs du menu,
   - titre du menu,
   - police de caractères à utiliser, etc...

   En outre, aucun contrôle n'est effectué pour vérifier l'existence de la police de caractères.

   Des primitives de plus haut niveau, effectuant tous ces contrôles et remplissant, sur base des arguments fournis, les records nécessaires, devraient être fournies tout en laissant la possibilité au programmeur d'utiliser les primitives de bas niveau si nécessaire. Ceci est d'ailleurs réalisé, par exemple, dans SUNVIEW structuré en plusieurs couches.

2) La gestion asynchrone des événements apporte l'avantage de ne pas monopoliser le serveur, mais il faudrait parfois pouvoir réaliser de la gestion synchrone d'événements. Reprenons l'exemple introduit à la section 2.10 à propos des fenêtres empilées. Une gestion synchrone peut résoudre ce problème ; le gestionnaire des entrées doit pouvoir dire au serveur qu'à l'enfoncement d'une touche de la souris, ce dernier cesse de traiter les événements jusqu'au moment où il reçoit une autorisation du gestionnaire des entrées.

3) Le réaffichage à charge de l'application en cas d'exposition d'une partie de fenêtre cachée, ne se justifie pas toujours. Il serait donc préférable que le serveur s'en occupe. Lorsque l'application a mis en place des structures de données dont elle peut profiter pour le réaffichage, ce dernier peut être pris en charge par l'application. On évite ainsi de dupliquer la même information qui serait nécessaire au serveur pour le réaffichage. Notons toutefois, que pour des raisons de performance, il est souvent préférable que le serveur s'occupe du réaffichage. En SUNVIEW, les choses se passent ainsi sans dégradation notoire de performance. La solution idéale serait de pouvoir préciser si le serveur doit ou non s'occuper du réaffichage.

4) La version 10 que nous avons utilisée manquait de robustesse et de tolérance aux erreurs.

2.13.2 Les points forts de X Windows

Au chapitre des points forts nous citerons :

1) l'efficacité et la bonne performance des primitives offertes,
2) la possibilité de travailler de manière transparente sur plusieurs écrans d'un réseau,
3) le statut de standard de fait de X Windows,
4) les concepts de haut niveau nombreux et appropriés pour la construction d'interfaces conviviales et attrayantes,
5) la documentation est complète, agréable à lire, bien illustrée et les primitives sont regroupées suivant les sections exposées de 2.3 à 2.12,
6) X Windows respecte les caractéristiques souhaitables pour un système de fenêtrage énoncées à la section 2.2.

2.14 Conclusion

Malgré certaines lacunes, X offre de nombreuses possibilités pour construire des interfaces à la fois attrayantes et performantes. Les primitives de base sont bien conçues et efficaces. Il est dommage que ces primitives soient de trop bas niveau et que des services qui pourraient être pris en charge par le serveur soient reportés sur l'application.

Chapitre 3

CEYX : un environnement de programmation générique et un langage orienté-objet

3.1 Introduction

Dans le système SACSO, toutes les manipulations d'arbres (arbre de syntaxe abstraite décrivant la représentation interne abstraite d'une spécification, ou structure intermédiaire décrivant la représentation externe concrète d'une spécification) sont écrites en CEYX.
Ce chapitre présente cette sur-couche de LE_LISP ainsi que quelques considérations qui permettent de la considérer comme un environnement de programmation indépendant du langage.

CEYX [Colnet 86a], [Colnet 86b], [Hullot 84] est un ensemble de primitives qui ajoute la notion d'objet au langage LE_LISP, dialecte du langage LISP [Chailloux 86], [Masini 84]. Ces primitives sont écrites en LE_LISP et s'utilisent comme des fonctions ordinaires : on parle dans ce cas d'une couche objet.
Il suffit de charger, dans l'environnement LE_LISP, les primitives LE_LISP qui réalisent la couche CEYX pour obtenir un interprète CEYX.

Dans un premier temps CEYX voulait, grâce à la notion de modèle, offrir au programmeur LE_LISP un mécanisme de structuration de données analogue aux records PASCAL ou aux structures C. CEYX devint ce qu'il est convenu d'appeler un langage orienté-objet par l'adjonction de relations hiérarchiques et de la notion d'héritage.

Mais CEYX est plus qu'un langage; c'est aussi un environnement de programmation indépendant du langage [Klint 83].

Avant de décrire CEYX comme langage et comme environnement en section 3.3, nous allons en premier lieu donner quelques notions de base concernant LE_LISP.

3.2 Notions de base concernant LE_LISP

Nous donnons un rapide aperçu du langage. Nous nous contenterons des notions indispensables pour une bonne compréhension de la suite.

3.2.1 Concepts importants

Les entités élémentaires manipulées par LE_LISP s'appellent des atomes. Ils sont répartis en trois catégories :

- les nombres (entiers ou réels)
- les chaînes de caractères

- les symboles qui sont pour l'instant des suites de caractères quelconques, exceptés quelques caractères qui ont une signification spéciale.

A tout objet LE_LISP est associé une valeur. Un atome numérique a pour valeur le nombre qu'il représente. Un atome symbolique a pour valeur l'objet LE_LISP qui lui est associé au moyen des fonctions SET ou SETQ.

Les atomes sont regroupés et combinés pour former des listes qui sont les structures fondamentales du langage. Une liste peut figurer comme élément d'une autre liste, entre parenthèses.

Il existe également des structures appelées vecteurs qui fournissent un accès indicé aux éléments d'un ensemble.

La programmation LE_LISP est de nature fonctionnelle : les instructions sont décrites par des fonctions qui s'appliquent à des arguments et délivrent un résultat. Les appels de fonctions se notent par des listes, selon un format préfixé (figure 3.2.1).

```
(+ 2 3)
(+ (* 3 4) (* 2 5))
```

Figure3.2.1 : format préfixé des appels de fonction

LE_LISP est un langage interprété. L'interprète évalue toute expression qui lui est soumise, puis imprime la valeur du résultat. Pour un appel de fonction, les arguments sont en général évalués séquentiellement de gauche à droite, puis la fonction est appliquée aux résultats de ces évaluations. Ce mécanisme est récursif.

La fonction "quote" permet de bloquer l'évaluation de son argument. Comme son usage est très fréquent, il est possible de l'abréger par une apostrophe (figure 3.2.2).

```
?(quote (+ 2 3))
=(+ 2 3)
?
?'(+ 2 3)
=(+ 2 3)
```

Figure3.2.2 : abbréviation de la fonction quote

3.2.1.1 Les symboles

Les symboles servent à nommer des fonctions mais aussi à désigner des valeurs. Les fonctions SET, SETQ servent à affecter une valeur à un symbole (figure 3.2.3). Un symbole

peut lui-même être valeur d'un autre symbole. Comme son nom l'indique, il sert alors à symboliser une entité sans qu'aucune valeur ne lui soit associée.

```
?(setq lieu 'Banque)
=Banque
?
?lieu
=Banque
?
?Banque
** eval : variable indéfinie : Banque
...
?
```

Figure 3.2.3 : utilisation des fonctions set et setq

L'évaluation de Banque provoque une erreur car aucune valeur n'est associée à ce symbole.
Lorsqu'un symbole apparaît en tête d'une liste à évaluer, il représente un nom de fonction. Par exemple , voici quelques appels de la fonction "list" qui retourne une liste construite avec les arguments de l'appel (figure 3.2.4).

```
?(list 1 2 3 4)
=(1 2 3 4)
?
?(list 'prune 'pomme 'poire)
=(prune pomme poire)
?
?(list 1 'prune 2 'pomme)
=(1 prune 2 pomme)
```

Figure3.2.4 : appels de la fonction list

Il est à noter qu'un même symbole peut désigner à la fois une fonction et une variable. C'est la définition de fonction qui est prise en compte si le symbole est en tête d'une liste à évaluer (figure 3.2.5).

```
?(list list 'bon 'roi 'Dagobert)
=(le bon roi Dagobert)
?
?(setq list 'le)
=le
?
?list
=le
?
```

Figure3.2.5 : symbole en tête ou pas de la liste

3.2.1.2 La représentation interne

La représentation interne des objets LE_LISP est indispensable pour apprendre à utiliser le langage au mieux de ses possibilités et pour comprendre certains effets de bord qui à priori peuvent porter à confusion. Toute expression est représentée en mémoire par un pointeur, c'est-à-dire une adresse vers sa ou ses valeurs.
L'accès aux valeurs d'un objet est toujours effectué par une indirection. Par conséquent, l'interprèteur est spécialisé dans la manipulation de pointeurs. Par exemple, une liste LE_LISP est représentée par un pointeur sur une liste chaînée de cellules physiques appelées cons. Dans chaque cellule figure un élément de la liste. Il existe une réserve de cons libre où l'interprète puise pour construire les listes, au fur et à mesure de ses besoins.

Une affectation consiste simplement à associer un symbole avec un pointeur sur la valeur affectée. Une même valeur peut donc être partagée par plusieurs objets différents, comme le montre l'exemple suivant (figure 3.2.6). La fonction "equal" teste si deux objets ont des valeurs égales tandis que la fonction "eq" teste s'ils ont même représentaton interne. La fonction "copy" permet de dupliquer physiquement une structure de données.

```
?(setq notes '(do re mi fa sol la si do))
=(do re mi fa sol la si do)
?(setq autres-notes notes)
=(do re mi fa sol la si do)
?;notes et autres-notes pointent maintenant
?;sur la même structure physique
?
?(setq encore-d-autres-notes (copy notes))
=(do re mi fa sol la si do)
?(equal notes encore-d-autes notes)
=t {true}
?(eq notes autres-notes)
  =t
```

```
?(eq notes encore-d-autres-notes)
=() {false}
?
```

Figure 3.2.6 : utilisation des fonctions equal et eq

Bien que leur implantation physique diffère, les valeurs des symboles sont les mêmes. Par conséquent, les tests avec le prédicat "equal" sont positifs. Ce n'est pas le cas avec le prédicat "eq" pour lequel deux expressions ne sont égales que si elles ont même représentation interne, c'est-à-dire même adresse.

3.3 Notions de base concernant CEYX

3.3.1 Introduction exemplative

Un record LE_LISP est défini à l'aide de la forme "defrecord" (figure 3.3.1).

(defrecord <nom> <champ1> ... <champN>).

Figure 3.3.1 : définition de la fonction defrecord

Cette construction associe au symbole <nom> la définition du Lrecord à N champs de nom <champs1>...<champsN>.

La fonction "omaked" permet de créer une instance d'un record (figure 3.3.2).

(omaked <nom-champs1>...<nom-champsN>)

Figure 3.3.2 : définition de la fonction omaked

L'accès aux champs peut se faire par les fonctions d'accès en lecture ou écriture engendrées automatiquement lors de la définition d'un record (figure 3.3.3), ou par la définition de constructions générales d'accès aux champs.

```
?(defrecord Individu nom prenom)
?(setq individu  (omaked Individu nom 'Dupond prenom 'Marcel))
?({Individu}:nom individu)
=Dupond
```

Figure 3.3.3 : fonctions d'accès aux champs d'un record

Supposons que nous voulions créer deux nouvelles structures possédant les deux champs du record Individu plus un nouveau champ "pointure" pour la première structure et un nouveau champ "condamnation" pour la seconde.

Une première solution consisterait à définir deux nouveaux records à trois champs. Pour simplifier la construction, CEYX introduit la notion de record extensible ou classe à l'aide de la fonction "defclass" (figures 3.3.4 et 3.3.5).

```
(defclass <nom>/{<superclasse>}:<nom> <champ1>...<champN>)
```

Figure 3.3.4 : définition de la fonction defclass

où <nom> devient le nom de la nouvelle classe ayant pour champs tous ceux de <superclasse> accessibles par les fonctions {<superclasse>}:<nom-du-champ>, et pour nouveaux champs tous les champs <champ...> accessibles par les fonctions {<nom>}: <nom-du-champ>; <nom> est sous-classe de <superclasse> qui est classe mère de <nom>.

```
?(defclass Individu nom prenom)
?(defclass {Individu}:Client pointure ).
?(defclass {Individu}:Suspect condamnation )
?(setq client  (omaked Client nom 'Dupond prenom 'Marcel  pointure 47))
?({Individu}:pointure client)
=47
```

Figure 3.3.5 : utilisation de la fonction defclass

Nous avons vu comment définir une structure de données, il nous reste à voir comment définir des opérations s'appliquant spécifiquement aux instances d'une telle structure. A la figure 3.3.6, nous définissons l'opération d'impression "print" pour les instances de la classe Individu. Cette opération est accessible soit par un appel de fonction LE_LISP en format préfixé, soit par invocation de la fonction "semcall". Auquel cas l'opération sera recherchée

préfixé, soit par invocation de la fonction "semcall". Auquel cas l'opération sera recherchée dans celles de Client, puis, en cas d'échec, dans les opérations des superclasses de Client, en l'occurence Individu.

```
?(de {Individu}:print (individu)
(print "nom :" ({Individu}:nom individu))
(print "prenom :" ({Individu}:prenom individu)))
?({Individu}:print client)
Nom: Dupond
Prenom: Marcel
=t
?(semcall 'Client 'print client)
Nom: Dupond
Prenom: Marcel
=t
?
```

Figure 3.3.6 : définition et appel d'une opération

Nous allons définir maintenant de nouvelles notions de record et de classe qui diffèrent des précédentes par le fait que leurs instances produites par la fonction "omaked" portent toujours l'information du record ou de la classe qui a permis de les engendrer. Ces nouveaux records sont appellés "les records taggés" (figure 3.3.7).

```
?(deftclass Individu nom prenom)
?(deftclass {Individu}:Client pointure ).
?(setq client  (omaked Client nom 'Dupond prenom 'Marcel pointure 47))
?
```

Figure 3.3.7 : utilisation des records taggés

Comme précédemment, nous pouvons sur un record taggé, définir une opération d'impression pour les instances de la classe Individu. Celle-ci peut être déclenchée soit par appel de fonctions LE_LISP, soit par invocation de la fonction "semcall" (figure 3.3.6).

Etant donné que les instances portent maintenant l'information du record ou de la classe qui a permis de les engendrer, nous pouvons donner une nouvelle construction "send" qui se différencie de "semcall" par le fait qu'elle ne prend plus pour argument le nom de record ou de classe (figure 3.3.8).

```
?(send 'print client)
Nom: Dupond
Prenom: Marcel
=t
?
```

Figure 3.3.8 : utilisation de la fonction send

La construction "send" permet donc d'effectuer élégamment des actions différentes suivant le type d'objet manipulé.

3.3.2 CEYX, un langage orienté-objet

3.3.2.1 Les modèles

La notion de modèle est une généralisation du record PASCAL. C'est un ensemble de champs de forme (nom du champ, type du champ (facultatif car il n'y a pas de contrôle de type), valeur par défaut).
Il est possible de choisir le mode d'implémentation de la structure : une table (vector), une liste de champs, un arbre de "cons", etc...

3.3.2.2 Les classes

En programmation orientée-objet, un programme est essentiellement un ensemble d'entités, appelées objets, auxquelles sont associées des opérations spécifiques.
Les ensembles d'objets ayant des comportements communs sont regroupés en classes. Une classe sert de moule pour la création des objets qui la représentent. Ce moule décrit une structure comprenant des variables, les champs, et des opérations associées, appelées méthodes.

Définir une classe d'un objet consiste à définir une structure de données et les procédures de manipulation pour cette structure. En CEYX, une classe est donc constituée d'un modèle et des méthodes associées.

3.3.2.3 L'instanciation

La classe est l'entité conceptuelle qui décrit tout objet lui appartenant. Une instance est un objet particulier construit en respectant les plans de construction de sa classe (fonction "omaked"). Elle possède les mêmes champs et mêmes méthodes que les autres instances de la classe, les champs prenant cependant les valeurs correspondant à la nature particulière de l'entité ainsi représentée.

3.3.2.4 L'héritage

Les classes sont organisées hiérarchiquement en une arborescence; la racine correspond à la classe générale (prédéfinie) dont toutes les autres sont issues. Les caractéristiques des classes supérieures sont héritées par les classes inférieures. Les champs du modèle et les méthodes de la classe mère (ou super-classe) sont adjoints aux champs et méthodes de chaque sous-classe. Ces champs hérités gardent leurs caractéristiques : type et valeur par défaut. Les méthodes automatiquement héritées peuvent être redéfinies pour la sous-classe, l'héritage étant ainsi inhibé.

3.3.2.5 La transmission de messages

Ce mécanisme permet, par le biais de la fonction "send", d'activer (ou lancer l'exécution) une méthode en fonction de la classe d'appartenance de l'objet qui reçoit le message, appelé receveur.
Le schéma général d'utilisation de la fonction "send" est décrit à la figure 3.3.9.

(send <message> <receveur> <suite...>)

Figure 3.3.9 : schéma générale d'utilisation de la fonction send

---

Le <message> est le nom (incomplet) de la méthode à activer. C'est en fonction de la classe d'appartenance du <receveur> que CEYX "complète" le nom de la méthode. Quand le nom est complet, CEYX provoque l'appel suivant (figure 3.3.10).

({???}:<message> <receveur> <suite...>)

Figure 3.3.10 : appel de la méthode correspondant au mesage

A la figure 3.3.10, la partie ??? représente le complément du nom de la méthode.

La partie <suite...> est optionnelle. Elle est utilisée quand la méthode à exécuter attend d'autres arguments que le receveur.

La différence entre un appel de fonction et une envoi de message est assez importante étant donné qu'on ne connait pas la fonction qui est exécutée à l'envoi du message sauf évidemment si nous connaissons le type du receveur.

(send + a b)

Figure 3.3.11 : envoi de message

A la figure 3.3.11, si a est du type entier alors la fonction + des entiers est exécutée, si a est du type complexe alors la fonction + des complexes est exécutée.

En plus de champs et de méthodes, un objet est pourvu d'une interface de communication qui permet de traiter les messages qui lui sont adressés. Cette interface comprend un dictionnaire de sélecteurs auxquels sont associées les procédures représentant les méthodes correspondantes. Par abus de langage, sélecteur de méthode et nom de méthode sont souvent confondus. Nous l'avons d'ailleurs fait jusqu'à présent. A la réception d'un message, l'objet recherche le sélecteur -le nom de la méthode- dans son dictionnaire, puis il exécute la procédure associée avant de transmettre le résultat au receveur. Lorsque le sélecteur ne figure pas dans le dictionnaire, il est recherché dans les dictionnaires des classes supérieures, en remontant dans l'arbre des classes par le lien de super-classe. La méthode utilisée pour répondre au message est fournie par le premier dictionnaire contenant le sélecteur concerné. Ce mécanisme correspond à l'héritage des méthodes.

Notons aussi que les méthodes qui sont des fonctions LE_LISP ordinaires peuvent être appelées directement sans passer par l'envoi de messages. Un appel direct à une méthode est plus rapide qu'un appel par l'intermédiaire de la fonction "send" car il supprime la phase de recherche de la méthode. En revanche, l'utilisation de send assure une plus grande évolutivité du programme.

3.3.2.6 L'autotypage

Pour permettre le mécanisme d'héritage, CEYX doit être capable de déterminer la classe d'appartenance des receveurs de message. CEYX peut mémoriser la classe d'appartenance dans cette instance qui est donc autotypée (cfr. les fonctions deftclass, et deftrecord).

3.3.2.7 Les packages

La réalisation du mécanisme d'héritage repose sur l'organisation hiérarchique des symboles LE_LISP à la manière des fichiers dans les systèmes d'exploitation de type UNIX. Ces symboles sont regroupés en packages, qui sont en quelque sorte l'équivalent des répertoires de ces systèmes. Un nom de symbole indique le chemin menant au package de l'identificateur utilisé. Le caractère ':' sert à séparer les noeuds composant le chemin. Le caractère '#' indique que nous partons de la racine de l'arborescence (le package global). Nous pouvons ajouter autant de sous-packages que nous voulons. Cette organisation arborescente permet également d'utiliser un même identificateur dans deux packages différents, ce qui est très utile pour éviter les conflits de noms entre symboles de classes différentes. Le mécanisme des packages a pour inconvénient d'allonger considérablement les noms de symboles. Pour les racourcir CEYX met à la disposition des utilisateurs un mécanisme d'abréviation. Outre

l'allégement de l'écriture, ce mécanisme permet de programmer une classe en ignorant dans quel package effectif elle se trouve. Il suffit de modifier la définition d'une abréviation pour déplacer une classe dans l'arbre des packages. L'abréviation du package est le nom de la classe. Par conséquent, nous pouvons faire le parallèle entre le mécanisme des abréviations et celui des classes.
Une notation entre accolades permet de désigner le package associé à la classe. Par exemple, pour associer une méthode à une classe, il faut définir une fonction LE_LISP dans le package de la classe. La notation entre accolades permet de désigner ce package.

3.3.3 CEYX, un environnement de programmation indépendant du langage

3.3.3.1 Introduction

L'implémentation d'un environnement de programmation dédicacé à un langage particulier demande un effort important de conception et de progammation si nous voulons l'adapter à un autre langage. Un environnement indépendant du langage, qui peut être automatiquement instancié à un langage particulier, permet un gain de temps considérable dans sa conception et son implémentation.
Un tel système offre également une interface uniforme, un programmeur qui utilise plusieurs langages différents peut travailler dans un environnement similaire pour ces différents langages.

CEYX est aussi un environnement de programmation indépendant du langage [Klint 83]. Le mot langage doit être pris au sens d'information hiérarchiquement structurée en générale. Un programme écrit dans un langage de programmation n'est qu'un cas particulier de ce genre d'information.

3.3.3.2 Structure générale d'un environnement de programmation indépendant du langage

Pour inclure un nouveau langage dans un environnement générique, dirigé par la syntaxe (figure 3.3.12), il faut habituellement fournir à cet environnement les définitions suivantes :

- *lasyntaxe lexicale :* celle-ci définit les mots clés et abréviations formant les terminaux du langage,

- *la syntaxe concrète :* celle-ci définit les règles de composition des terminaux du langage qui doivent être respectées pour former des expressions syntaxiquement correctes dans ce langage,

- *la syntaxe abstraite :* celle-ci définit les règles de formation d'arbres abstraits syntaxiquement corrects qui seront utilisés pour manipuler les expressions de ce langage,

- *la sémantique statique* : celle-ci définit les contraintes qui doivent être vérifiées sur les expressions bien formées. Dans le cas d'un langage de programmation, il s'agit des contraintes habituellement vérifiables par un compilateur (déclarations des variables, contraintes de type, ...). Ces contraintes sont en fait d'ordre syntaxique, mais elles sont dépendantes du contexte,

- *la sémantique dynamique* : celle-ci définit la signification des expressions. La définition est souvent donnée de façon procédurale. Dans le cas d'un langage de programmation, il s'agit de l'exécution d'un programme écrit dans ce langage.

Toutes ces définitions (sauf celle de la sémantique dynamique), sont introduites grâce à un formalisme. Celui-ci a généralement une structure de type BNF.

Le noyau de l'environnement est composé d'outils permettant le dialogue dans le langage défini. Les principaux outils sont :

- *l'analyseur lexical et syntaxique :* celui-ci reçoit une expression sous sa forme textuelle externe et fournit grâce au constructeur d'arbres de syntaxe abstraite, l'arbre abstrait correspondant à l'expression ; selon le langage défini,

- *le décompilateur :* celui-ci fournit une représentation textuelle à partir de l'arbre abstrait d'une expression. Le décompilateur du langage, est construit à partir des "fonctions de décompilation" qui sont le lien logique entre la forme arborescente (ou abstraite) et la forme textuelle (ou concrète),

- *l'éditeur syntaxique :* celui-ci permet la construction et la modification d'expressions syntaxiquement correctes manipulées sous leur forme arborescente abstraite.

Les définitions de la syntaxe concrète et lexicale contiennent suffisamment d'informations que pour créer un analyseur lexical et syntaxique pour le nouveau langage défini.
L'arbre syntaxique contient typiquement des non-terminaux (<expression>) au niveau de ses noeuds et des terminaux (mots clés) au niveau de ses feuilles. Cet arbre syntaxique est transformé en un arbre de syntaxe abstraite. Celui-ci contient typiquement des notions sémantiques (while-statement, plus-operator, ...) au niveau de ses noeuds et des constantes et identifieurs au niveau de ses feuilles. Un noeud d'arbre abstrait va représenter un concept représentatif (une idée) pour l'utilisateur. Ce concept peut lui-même être composé d'autres concepts qui constitueront les noeuds fils, et ainsi de suite. Les mots clé de la syntaxe concrète n'apparaissent pas dans les règles définissant la syntaxe abstraite. A tout concept représentatif pour l'utilisateur que nous trouvons dans le langage, il y correspond un opérateur (ou constructeur) de syntaxe abstraite. Un tel opérateur est défini par une règle de syntaxe abstraite qui précise quels sont les opérateurs pouvant apparaître légitimement comme fils de cet opérateur dans l'arbre abstrait.

Durant une édition dirigée par la syntaxe du langage, l'arbre abstrait d'un programme est construit directement et indépendamment de l'arbre syntaxique.

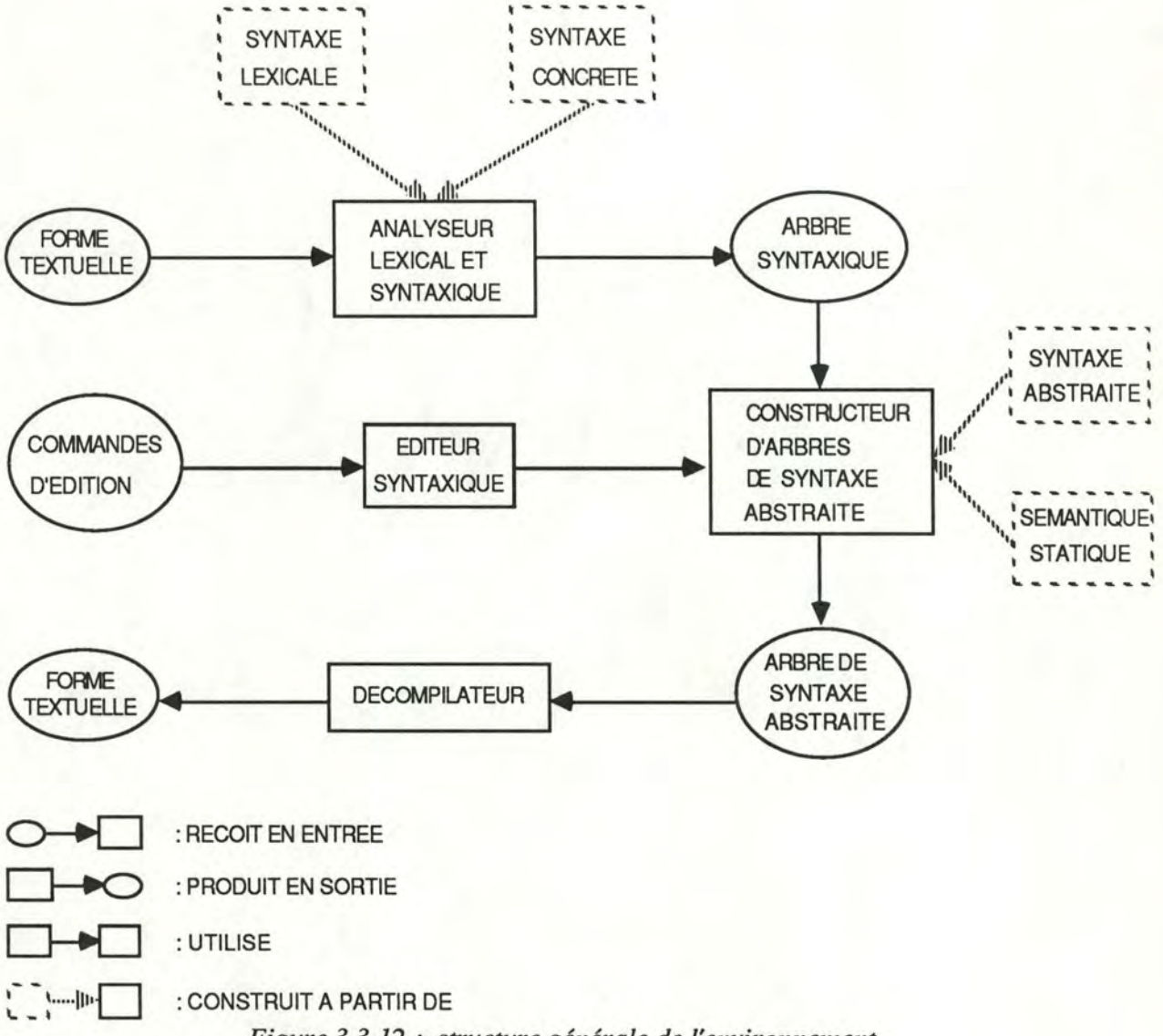


Figure 3.3.12 : structure générale de l'environnement

3.3.3.3 Exemple pour l'analyse

Dans les sous-sections suivantes, nous reprenons de [Klint 83], quelques exemples de l'environnement de programmation développé sous CEYX pour le langage de programmation PICO. La syntaxe du langage PICO sous forme B.N.F. est la suivante :

```
<pico-program>    ::=  'program' <decls> <series> 'end'.
<decls>           ::=  'declare' <id-list> ';'.
<id-list>         ::=  <id-list> ',' <id>  |  <id>.
<series>          ::=  <series> ';' <statement> | <statement> .
<statement>       ::=  <asg-stat> | <if-stat> | <while-stat>.
<asg-stat>        ::=  <id> ':=' <exp>.
<if-stat>         ::=  'if' <exp> 'then' <series>
                       'else' <exp> 'fi'.
<while-stat>      ::=  'while' <exp> 'do' <series> 'od'.
```

<exp> ::= <simple-exp> '+' <simple-exp> |
<simple-exp> '*' <simple-exp> |
<simple-exp>.
<simple-exp> ::= <id> | <number> | '(' <exp> ')'.

avec <id>, et <number> qui sont des constantes entières.

3.3.3.4 Propriétés de CEYX par rapport à cette structure générale

CEYX ne supporte pas la notion de **sémantique statique**. Tous les tests vérifiant les contraintes de sémantique statique doivent être programmés explicitement en LE_LISP.
CEYX ne permet pas de définir les **syntaxes lexicale** et **concrète** d'un langage. CEYX ne permet donc que l'édition d'un programme dirigée par la syntaxe du langage. Il n'offre pas de moyen de passer d'une forme textuelle à une forme arborescente (cfr. l'analyseur lexical et syntaxique à la figure 3.3.12). Par contre, à l'aide des concepts de **constructeur** et d'**univers**, CEYX permet de définir la **syntaxe abstraite** du langage. Plus précisément le concept d'univers de CEYX a les propriétés d'une définition complète d'un langage : il permet de définir la syntaxe abstraite du langage, celle-ci donnant les règles de dérivation d'arbres abstraits syntaxiquement corrects pour ce langage.

En CEYX, il est possible de manipuler un arbre de syntaxe abstraite dans lequel les sous-arbres peuvent appartenir à des formalismes différents. Lors d'une édition dirigée par la syntaxe, le passage d'un formalisme à un autre se fait en accédant, au niveau de l'arbre de syntaxe abstraite, à un noeud univers différent.

Un exemple de l'environnement de programmation développé sous CEYX pour le langage PICO, clarifie ces concepts de constructeurs et d'univers (figure 3.3.13).

```
(defuniverse pico)
(defcons program~pico (decls series))
(defcons decls~pico id)
(defcons series~pico statement)
(defuniverse statement~pico)
...
(defcons while-stat~statement (exp statement))
(defuniverse exp~pico)
(defcons plus~exp (exp exp))
...
```

Figure 3.3.13 : les concepts de constructeur et d'univers

Le nouvel univers est tout d'abord défini. Les trois constructeurs program, decls, series qui appartiennent tous à l'univers pico, sont ensuite définis.

CEYX permet également la définition de **la sémantique dynamique** du langage. Celle-ci s'exprime sous forme de procédures qui a l'exécution décrivent la sémantique du langage.

Une sémantique dynamique peut être associée avec un noeud de l'arbre abstrait. En général, il suffit d'associer une fonction sémantique sous un certain nom avec un constructeur ou un univers dans l'arbre.

Dans l'exemple suivant (figure 3.3.14), nous associons la propriété sémantique propsem avec le constructeur constr :

```
(defsem (constr propsem) objet corps)
```

Figure 3.3.14 : définition d'une propriété sémantique

L'objet représente les arguments du code LE_LISP représenté par le corps, qu'il faut exécuter quand la propriété propsem est invoquée via une notation qui associe une clé à une propriété sémantique (cfr. 3.3.3.5 Interface utilisateur et primitives de base). La définition d'une propriété sémantique est donnée à la figure 3.3.15.

La **décompilation** consiste à créer une image à afficher à partir de l'arbre de syntaxe abstraite. CEYX traverse cet arbre de syntaxe abstraite dans un préordre descendant récursif. Pendant cette décompilation, le **décompilateur** doit maintenir un niveau d'indentation qui peut être incrémenté ou décrémenté au début de la décompilation de certaines constructions. Le décompilateurr doit alors résoudre deux problèmes :

1) il doit choisir le format correct pour la décompilation de ces constructions. Prenons par exemple, le cas d'une construction "if-stat" avec une partie "else" qui est vide alors que cette construction est incluse dans une autre construction "if-stat",

2) il doit savoir que faire lorsque le décompilateur dépasse la ligne courante.
   En principe, il existe deux solutions :

   - soit associer plusieurs formats de décompilation à chaque constructeur et choisir le format en fonction de l'espace disponible et de l'information contenue dans l'arbre abstrait,

   - soit n'associer qu'un seul format à chaque constructeur mais permettre une interprétation variable de ce format en fonction de l'espace disponible. CEYX fournit une notion formelle pour la définition de formats à multiples interprétations. La méthode de spécification de la décompilation est basée sur les notions de blocs de texte horizontal, et vertical. Ces blocs de texte peuvent contenir explicitement des points de passage à la ligne. Le morceau de texte placé entre deux points de passage à la ligne forme le composant du bloc. Les composants d'un bloc horizontal sont placés de gauche à droite sur la ligne courante. Si la ligne courante dépasse l'espace disponible,

un point de passage à la ligne est créé commençant à une indentation donnée. Après quoi les composants restants sont placés. Ce processus est répété tant que tous les composants du bloc ne sont pas affichés. Les composants d'un bloc vertical sont placés sur la même ligne horizontale lorsqu'il y a assez d'espace, sinon sur des lignes consécutives avec le même montant d'indentation.

Nous donnons un exemple dans lequel la propriété sémantique pretty est associée aux constructeurs program et while-stat (figure 3.3.15).

```
(defsem (program pretty) (x)
  (vterpri)
  (vpatom "program")
  (vblock-with-indent 4
        (cutpoint)
        (vdispatch (get-son x 1)) ; partie decls
        (vpatom ";")
        (cutpoint)
        (vdispatch (get-son x 2)) ; partie series
  )
  (vterpri)
  (vpatom "end")
)

(defsem (while-stat pretty) (x)
  (vblock-with-indent 0
        (hblock
              (vpatom "while")
              (cutpoint)
              (vdispatch (get-son x 1))
        ) ; partie test
        (cutpoint)
        (vblock
              (vpatom "do") (cutpoint)
              (vdispatch (get-son x 2))
        ) ; partie do
        (cutpoint)
        (vatom "od")
  )
)
```

Figure 3.3.15 : définition d'une propriété sémantique

A la figure 3.3.15 les fonctions en caractères gras sont des fonctions de décompilation :

- vblock (hblock) définit un block vertical (horizontal)
- vblock-with-indent N (hblock-...) définit explicitement l'indentation pour le block
- vpatom affiche un atome
- vterpri réalise un passage à la ligne
- cutpoint définit un point de passage à la ligne
- vdispatch décompile récursivement les sous-arbres.

3.3.3.5 Propriétés de l'environnement de programmation CEYX

1) Interface utilisateur et primitives de base

CEYX permet une définition dynamique de "clés" qui définissent les propriétés de l'interface utilisateur. A chaque moment, pendant l'édition, il existe un objet courant. Précédemment, nous avons vu qu'à un objet nous pouvions associer des propriétés sémantiques. Ces propriétés sémantiques peuvent être invoquées via un notation qui associe clé et proprité sémantique. Pendant l'édition, l'utilisateur utilise ces clés comme des commandes du système. L'interprétation de ces clés dépend de leur définition dans l'objet courant. L'interprétation des clés est complètement dynamique : si l'objet courant ne contient pas la définition d'une certaine clé, la définition de la clé est recherchée dans l'objet parent et ainsi de suite jusqu'à ce qu'une définition soit trouvée. Si aucune définition n'est trouvée le système fournit une solution par défaut ou un message d'erreur. Ainsi les opérations standards d'édition comme "descendre", "monter", etc. peuvent être spécialisées pour certaines classes d'objets. Il suffit d'associer à l'objet concerné une expression LE_LISP réalisant cette définition de clé.

2) Communication extérieure

CEYX permet de sauver des arbres abstraits sur fichier externe en sauvant une expression LE_LISP qui recrée l'arbre.

3) Annotations

CEYX permet la manipulation simultanée d'arbres abstraits appartenant à différents langages. Ceci permet, par exemple, la manipulation de programmes PASCAL dans lesquels certains composants sont écrits en un langage de spécification formel.

3.4 Conclusion

CEYX est un environnement de programmation :

- multi-langages,
- qui offre une implémentation de types abstraits : des structures de données sont définies par le biais de fonctions de manipulation (modèle et méthode),
- qui offre une programmation orientée objet (à la Smalltalk) : envoi de message, héritage de champs et de fonctions.

Les avantages des environnements de programmation indépendants du langage se mesurent, non seulement dans le gain de temps au niveau des efforts de conception et de programmation, mais également en termes de généralité et de sophistication du résultat.
Il reste cependant beaucoup à faire pour la construction d'un environnement de programmation qui couvre le cycle complet d'édition-exécution-debugging et qui supporte une méthodologie explicite pour le développement de logiciel.

PARTIE II

REALISATION D'UN OUTIL DE CONSTRUCTION D'INTERFACES INTERACTIVES

Chapitre 4

Modification du multi-fenêtrage de SACSO

4.1 Introduction

SACSO était installé sur SM90 avec écran alphanumérique. Suite à son portage sur station de travail, il est apparu que le gestionnaire de multi-fenêtrage écrit en LE_LISP ne profitait pas des possibilités de l'écran bitmap d'une station. Une modification du multi-fenêtrage s'avérait donc utile. Le projet que nous avions à réaliser se situait manifestement dans la phase de maintenance et plus précisément d'extension du projet. Cette phase de maintenance peut en fait se découper suivant les différentes étapes précédentes du cycle de vie appliquées à un sous-ensemble ou une nouvelle partie du produit logiciel.

Ce chapitre décrit les différentes étapes de construction du nouveau système de multi-fenêtrage. Remarquons d'abord que nous n'utilisons pas le terme "multi-fenêtrage" au sens de découpe de l'écran en terminaux virtuels avec un processus attaché à chaque terminal virtuel. Il s'agit simplement de découper l'écran en fenêtres.

Avant d'aborder en section 4.3 les différentes étapes de la construction du nouveau multi-fenêtrage, nous donnons, en section 4.2, quelques critères d'évaluation des interfaces homme-machine qui nous ont conduits dans la conception du nouveau multi-fenêtrage de SACSO.

4.2 Critères d'évaluation d'une interface homme-machine

Le développement d'interfaces homme-machine devient un besoin vital. Le critère d'évaluation qui importe pour l'utilisateur est bien sûr la convivialité [Schneiderman 87]. Ce critère est cependant très difficile à évaluer et nécessite l'intervention de spécialistes dans des domaines aussi divers que l'ergonomie et la psychologie cognitive. Schneiderman à cet effet énonce le principe de **reconnaissance de la diversité,** c'est-à-dire que l'interface doit pouvoir tenir compte du profil de l'utilisateur. L'interface ne doit pas seulement être conçu pour une seule classe d'utilisateurs comme par exemple des experts mais aussi pour des novices et des utilisateurs occasionnels. Pour cela différents styles d'interaction avec l'utilisateur sont enviseagables comme par exemple les menus, les commandes ou la manipulation directe.
Schneiderman propose huit règles qui devraient être suivies lors de la conception d'une interface :

1) veiller à la **cohérence** du dialogue
   exemple : pour effectuer une même action, il faut toujours pouvoir utiliser la même commande quel que soit le niveau où l'on se trouve
2) offrir la possibilité aux utilisateurs avertis d'utiliser des commandes plus puissantes, des "**raccourcis**",
3) fournir un "**feedback**" explicite à une action (lorsqu'une action est exécutée, il faut offrir une information en retour),
4) organiser le dialogue en **sessions**, avec début milieu et fin, afin d'offrir à l'utilisateur la satisfaction d'avoir terminé la session,

5) offrir un traitement d'erreur simple c'est-à-dire que l'utilisateur doit pouvoir réparer de manière simple une erreur qu'il a commise,
6) permettre la **réversibilité des actions** c'est-à-dire la possibilité de défaire, d'annuler les effets d'une action,
7) permettre un contrôle de l'interface par l'utilisateur, de manière à ce qu'il ait le sentiment d'être l'initiateur du dialogue,
8) réduire la charge de la mémoire à court terme par exemple en rappelant à l'utilisateur les commandes qu'il peut effectuer.

Schneiderman propose pour évaluer le caractère convivial d'une interface, les critères suivants :

- temps d'apprentissage,
- rapidité des interactions, réactions de l'interface (bien que dans certains cas un temps de réponse trop rapide s'avère mauvais),
- taux d'erreur des utilisateurs,
- satisfaction subjective de l'utilisateur,
- degré de mémorisation par l'utilisateur, des informations nécessaires à l'utilisation de l'interface.

Ces facteurs ont plus ou moins d'importance suivant le type de système à concevoir. Dans le cas de systèmes critiques tels des systèmes de contrôle du traffic aérien ou de pilotage de centrales nucléaires, l'accent est mis sur la rapidité de réponse du système, et sur le taux d'erreur faible de l'utilisateur même si c'est au prix d'une perte de satisfaction de l'utilisateur ou d'un temps d'apprentissage plus long.

4.3 Les différentes étapes de la modification du système de multi-fenêtrage

4.3.1 Introduction

Nous avons dans la mesure du possible découpé le travail à réaliser suivant les différentes étapes du cycle de vie [VanLamsweerde 85]. Etant donné la taille du projet à réaliser et le fait que l'on se situe dans une phase de maintenance, certaines étapes seront moins développées ou même passées sous silence.

Il est important de signaler que SACSO a été conçu dans une optique de prototype et dans un cadre de recherche. Sa construction a donc été faite de manière incrémentale.

4.3.2 Première étape : analyse des besoins

4.3.2.1 Les "problèmes"

La nécessité de remplacer le gestionnaire de multi-fenêtrage existant nous paraît justifié pour les raisons suivantes :

- SACSO conçu initialement pour tourner sur SM90 a été porté sur un SUN qui dispose d'un écran bitmap. De ce fait le multi-fenêtrage existant ne profite pas des possibilités plus étendues du SUN ne serait-ce qu'au niveau de la taille de l'écran, de l'utilisation

de la souris et des possibilités graphiques. Ainsi, la sélection d'une entité d'information dans une fenêtre ou un menu, se fait grâce à des touches de contrôle pour déplacer le curseur.
L'utilisation de la souris pour cette sélection serait plus rapide. C'est surtout au niveau de l'interaction avec l'utilisateur que l'interface pourrait être améliorée grâce à l'utilisation de la souris,

- le multi-fenêtrage écrit en LE_LISP est bien conçu, mais il a toutefois un mauvais temps de réponse et a tendance à fruster l'utilisateur qui se demande si le système a ou non reçu sa commande.

4.3.2.2 Les solutions

La solution pour améliorer le multi-fenêtrage actuel est de construire un **nouveau gestionnaire de multi-fenêtrage** et de l'intégrer à SACSO. Ce nouveau gestionnaire est responsable de la gestion des objets graphiques (fenêtre, menu, ...) à l'écran et de la gestion des entrées (clavier, souris). Les objets graphiques pouvant se recouvrir à l'écran, nous parlerons dans la suite d'une pile d'objets graphiques pour désigner l'ensemble des objets graphiques affichés à l'écran. Les fonctionnalités attendues du gestionnaire de multi-fenêtrage par l'utilisateur et le programmeur sont différentes. Du point de vue de l'utilisateur, le gestionnaire de multi-fenêtrage doit permettre :

- de détruire des objets graphiques,
- de redimensionner des objets graphiques,
- de déplacer des objets graphiques,
- de placer un objet graphique au sommet (bas) de la pile des objets graphiques,
- d'introduire du texte,
- de sélectionner un objet ou composant d'un objet graphique au moyen d'un dispositif de désignation.

Du point de vue du programmeur chargé de construire une interface grâce au gestionnaire de multi-fenêtrage, ce dernier doit permettre en plus :

- de créér des objets graphiques,
- d'afficher des objets graphiques,
- d'obtenir la sélection faite par l'utilisateur au moyen du dispositif de désignation,
- d'obtenir le texte introduit par l'utilisateur.

En premier lieu, nous avons recensé les outils à notre disposition pour construire un tel gestionnaire. Ensuite nous avons évalué les possibilités des différents outils disponibles afin de justifier notre choix.

Cette liste n'est évidemment pas exhaustive dans la mesure où d'autres gestionnaires sont disponibles sur le marché, mais non disponibles au CRIN au moment de notre stage. D'autre part, nous ne nous sommes pas fixés pour objectif de réaliser une étude complète des outils spécialisés pour construire des systèmes de multi-fenêtrage. Nos critiques concernant les différents systèmes sont émises suite à des lectures personnelles et des renseignements obtenus

ci et là auprès du personnel du CRIN. Une étude approfondie se serait certainement avérée bénéfique mais n'a pu être engagée vu le temps que nous aurions dû y consacrer.

Les systèmes de multi-fenêtrage disponibles au CRIN sont les suivants :
- Sunview disponible sur SUN MICROSYSTEMS [Sun 87],
- X Window System (X Windows) [XWindows 87],
- Le-Tool développé par l'INRIA [Chailloux 86].

Les avantages et inconvénients de ces systèmes sont les suivants :

a) **Sunview**

- Avantages :
  - Sunview est disponible sur le SUN,
  - la documentation fournie est bonne.
  - un interfaçage est possible avec LE_LISP via des modules écrits en langage C et utilisant les primitives de Sunview,
  - Sunview est déjà utilisé par des personnes au CRIN.

- Inconvénients :
  - Sunview manque de portabilité du fait qu'il n'est disponible que sur SUN.

b) **X Window System** (X Windows)

- Avantages :
  - X Windows a tendance à devenir un standard de facto,
  - il est portable (X Windows n'est pas développé par un constructeur unique).

- Inconvénients :
  - X Windows n'est pas encore utilisé au CRIN,
  - la version disponible est une version test sans documentation,
  - il n'offre que des primitives graphiques de bas niveau.

c) **Le-Tool**

- Avantages :
  - Le-Tool est en fait une interface LE_LISP-C-SUNTOOL et peut donc être utilisé directement en LE_LISP,
  - nous avons réalisé un mini-prototype écrit en Le-Tool rencontrant la faisabilité de ce choix.

- Inconvénients :
  - la documentation disponible est insuffisante,
  - pour lancer Le-Tool il faut nécessairement entrer dans Suntools qui continue a exercer un contrôle sur les objets créés par Le-Tool.

4.3.2.3 Choix de la solution

Eu égard à ce qui précède le choix du système s'est porté sur l'ensemble d'outils proposés par X Windows. Ce choix est motivé par les outils disponibles au CRIN au moment du choix. Nous avons choisi X Windows principalement pour des raisons de portabilité et de standard. Cependant, vu que les primitives offertes par X Windows sont de bas niveau, nous avons décidé de construire une couche au-dessus de X Windows. Cette couche sera présentée à la section 4.3.5.3.

4.3.2.4 Définition générale des objectifs du projet

Les objectifs poursuivis pour les modifications apportées à SACSO visent à améliorer SACSO au niveau de :

- la facilité d'utilisation,
- l'interface homme-machine.

4.3.3 Deuxième étape : spécification fonctionnelle

4.3.3.1 Introduction

Nous n'allons pas réaliser à proprement parler une analyse fonctionnelle puisque nous nous situons dans une phase de maintenance et que nous n'allons procéder qu'à des remplacements de modules (cfr 4.3.4.2). Une spécification de l'interface usager que nous allons réaliser nous paraît tout de même nécessaire. Pour la conception de cette interface, nous avons tenu compte des recommandations de Schneiderman énoncées à la section 4.2, qui s'avèrent être un excellent guide.

4.3.3.2 Spécification du nouveau gestionnaire de multi-fenêtrage de SACSO

Cette section décrit la spécification en SACSO du gestionnaire de multi-fenêtrage à construire. Le système SACSO est décrit comme un ensemble à deux composants :

- le gestionnaire de multi-fenêtrage (cfr 4.3.2.2) qui s'occupe de recevoir toutes les commandes de l'utilisateur qu'il peut prendre en charge, comme par exemple déplacer un objet graphique interactif,
- l'application, qui s'occupe des commandes de l'utilisateur qui lui sont destinées. L'application ne sera pas spécifiée dans cette section.

Entre ces deux composants, il y a la file de l'application dans laquelle le gestionnaire de multi-fenêtrage place les commandes de l'utilisateur dont il ne peut s'occuper. L'application se contente de lire les commandes de l'utilisateur qui lui sont destinées dans cette file et d'y répondre. L'application, quant à elle, place dans la file du gestionnaire de multi-fenêtrage les actions que celui-ci doit réaliser (exemple : créer une fenêtre, ...). Seul le composant "gestionnaire de multi-fenêtrage" sera spécifié. Cette architecture simplifiée est présentée à la figure 4.3.1.

Du point de vue de l'utilisateur, un gestionnaire de multi-fenêtrage doit permettre de déplacer des objets graphiques interactifs (menu, fenêtre, ...), de les faire apparaître, disparaître etc... (cfr 4.3.2.2).

Nous allons d'abord commencer cette spécification par la description des types utilisés, selon une stratégie purement descendante. Pour la signification des opérations prédéfinies utilisées, on consultera l'annexe 3.

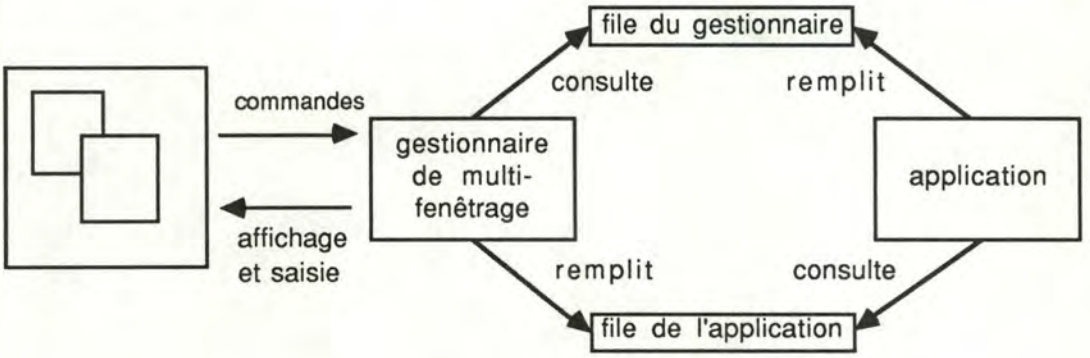


Figure 4.3.1 : architecture simplifiée du sytème SACSO

TYPE **MULTIFENETRAGE**

Partie lexique

But : un système de multi-fenêtrage est composé :

- d'un écran (écran) servant à afficher l'information,
- d'une mémoire (mémoire) pour stocker toute l'information à afficher à l'écran,
- d'une suite d'objets graphiques (objgraph) affichés à l'écran.

Partie formelle

Structure : PC [écran : **ECRAN**, mémoire : **MEMOIRE**,
objgraph : **OBJS-GRAPHS**]

TYPE **ECRAN**

Partie lexique

But : un écran est composé

- d'une taille d'écran (taille-ec),
- d'un affichage représentant l'information affichée à l'écran (affichage).

Partie formelle

Structure : PC[taille-ec : **POSITION-ECRAN,**
affichage : T[**POSITION-ECRAN, INFORMATION**]]

TYPE **MEMOIRE**

Partie lexique

But : une mémoire est composée de tampons. Chaque tampon a un nom unique et est associé à un objet graphique. Il contient l'information à afficher dans l'objet graphique.

Partie formelle

Structure : T[**NOM-TAMPON, TAMPON**]

TYPE **OBJS-GRAPHS**

Partie lexique

But : liste de tous les objets graphiques affichés à l'écran. Les objets graphiques pouvant se recouvrir à l'écran, on parlera d'une pile d'objets graphiques affichés à l'écran. L'ordre des objets graphiques dans la liste définit l'ordre dans lequel les objets sont affichés à l'écran. Cet ordre est défini comme suit :

- le premier objet dans la liste se trouve au sommet (position 1) de la pile des objets graphiques ; il est entièrement visible,
- le $i^{ème}$ ($2 \leq i \leq$ taille (liste) -1) objet dans la liste est à la $i^{ème}$ position dans la pile des objets graphiques ; il peut être caché en partie ou en totalité par un autre objet graphique,
- le dernier objet dans la liste se trouve au bas (position taille (liste)) de la pile des objets graphiques ; il peut être caché en partie ou en totalité par un autre objet graphique.

Partie formelle

Structure : S[**OBJ-GRAPH**]

TYPE **OBJ-GRAPH**

Partie lexique

But : un objet graphique interactif est composé

- d'une position à laquelle il est affiché à l'écran (pos-écran),
- d'une taille, composée de la largeur et de la hauteur de l'objet graphique (taille-objet),
- d'un nom de tampon ; ce tampon contient l'information à afficher dans l'objet graphique (nom-tampon),
- d'une position dans le tampon à partir de laquelle l'information dans le tampon est affichée dans l'objet graphique (pos-tampon),
- d'un identificateur d'objet graphique (id-objet).

Partie formelle

Opérations associées : DEPLACER, MONTER, DESCENDRE, DETRUIRE,
SELECTION

Structure : PC [pos-écran : **POSITION-ECRAN** ,
taille-objet : **POSITION-ECRAN**,
nom-tampon : **NOM-TAMPON**,
pos-tampon : **POSITION**,
id-objet : **ENTIER**]

TYPE **TAMPON**

Partie lexique

But : à chaque objet graphique est associé un tampon qui contient tout ce qui doit être affiché dans l'objet graphique. Un tampon est composé d'une taille de tampon (taille-tampon) et d'un contenu de tampon (contenu). La taille de tampon est composée de la largeur et de la hauteur du tampon.

Partie formelle

Structure : PC[taille-tampon : **POSITION**,
contenu : T[**POSITION, INFORMATION**]]

TYPE **POSITION**

Partie lexique

But : une position est composée d'un numéro de ligne (axe des y) et d'un numéro de colonne (axe des x).

Invariant : une position ne peut avoir ses deux composantes nulles.

Partie formelle

Structure : PC [colonne : **ENTIER**, ligne : **ENTIER**]

Invariant : (colonne (pos) supérieur 0) et (ligne (pos) supérieur 0)
avec pos : **POSITION**

TYPE **POSITION-ECRAN**

Partie lexique

But : une position-écran est composée d'un numéro de ligne (axe des y) et d'un numéro de colonne (axe des x)

Invariant : une position-écran ne peut avoir ses deux composantes nulles, ni la composante ligne supérieure à la hauteur de l'écran, ni la composante colonne supérieure à la largeur de l'écran.

Partie formelle

Structure : **POSITION**

Invariant : ((colonne (pos) supérieur 0) et (ligne (pos) supérieur 0))
et (((colonne (pos) infeg colonne (taille-ec (écran))
et (ligne (pos) infeg ligne (taille-ec (écran)))))
avec écran : **ECRAN**, pos : **POSITION-ECRAN**

TYPE **INFORMATION**

Partie lexique

But : décrit le type d'information affichée à l'écran et contenue dans les différents tampons des objets graphiques

Partie formelle

Structure : U [**CAR, CAR-GRAPH**]

TYPE **CAR-GRAPH**

Partie lexique

But : décrit un caractère graphique.

Partie formelle

Structure : S[**ENTIER**]

TYPE **NOM-TAMPON**

Partie lexique

But : nom d'un tampon

Partie formelle

Structure : S [**CAR**]

TYPE **FILE-APPLICATION**

Partie lexique

But : une file de l'application est une liste dont chaque élément est composé

- d'un identificateur de l'objet sur lequel une action a été effectuée par l'utilisateur (id-objet),
- d'une information sélectionnée par l'utilisateur pour réaliser une action (information).

Partie formelle

Structure : S[PC[id-objet : **ENTIER**, information : S[**INFORMATION**]]]

En guise de récapitulatif, nous donnons à la figure 4.3.2 la forêt des types de la spécification MULTIFENETRAGE.

Nous allons maintenant décrire les opérations qu'un utilisateur peut effectuer sur un objet graphique interactif.

Opération DEPLACER :

MULTIFENETRAGE, POSITION-ECRAN, OBJ-GRAPH
-> MULTIFENETRAGE

Partie lexique

But : déplacer un objet graphique (obj-graph) et son contenu à une position pos à l'écran. Déplacer un objet le fait apparaître entièrement à l'écran, au sommet de la pile des objets graphiques. Pour déplacer un objet les opérations suivantes sont effectuées :

- calculer sa nouvelle postion à l'écran et vérifier que cette postion est toujours dans l'écran (ligne (pos-ec') et colonne (pos-ec')),
- le déplacer au début de la liste des objets graphiques objs-graphs,
- il faut effacer la zone d'écran (appelée par la suite zone A) occupée par l'objet à la position qu'il occupe avant l'application de l'opération (effacer_zone),
- il faut effacer la zone d'écran (appelée par la suite zone B) qui sera occupée par l'objet à la position à laquelle il sera déplacé (effacer_zone),
- il faut réafficher le contenu de l'objet dans la zone B (afficher_zone_tampon),
- enfin, il faut réafficher les parties ou totalité de contenu d'objets rendus visibles dans la zone A suite au déplacement de l'objet (réafficher_zone).

Objets :

mult, mult' : **MULTIFENETRAGE**
ec, ec', ec", ec"', ec"" : **ECRAN**
obj-graph, obj-graph' : **OBJ-GRAPH**
i, j : **ENTIER**
mem : **MEMOIRE**
pos-ec, pos-ec', taille-obj : **POSITION-ECRAN**
nom-tamp : **NOM-TAMPON**
pos-tamp : **POSITION**
objs-graphs, objs-graphs' : **OBJS-GRAPHS**
pos : **POSITION-ECRAN**

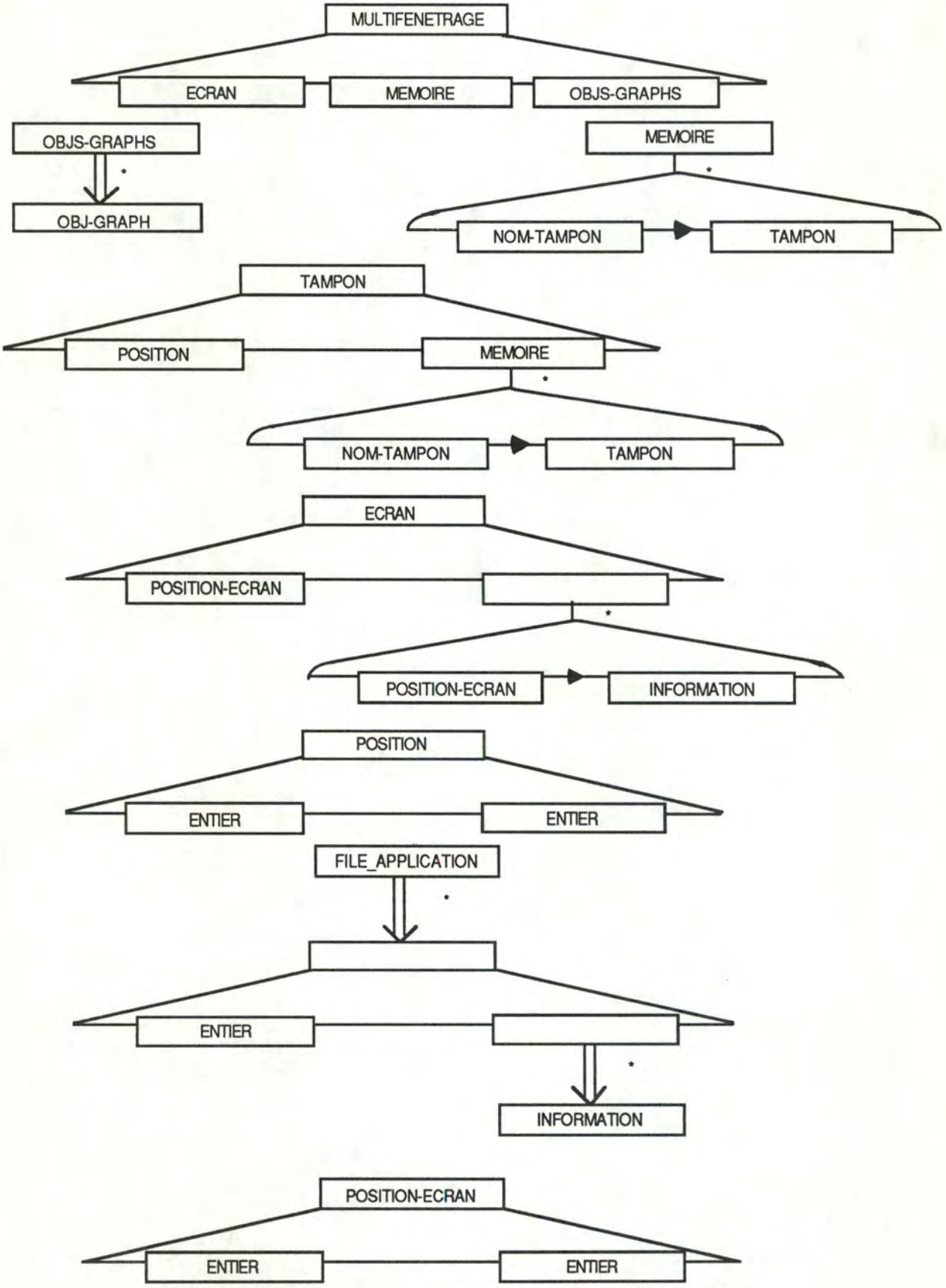


Figure 4.3.2 : forêt des types de la spécification multi-fenêtrage

Partie formelle

Définition :

mult' = DEPLACER (mult, pos, obj-graph)

avec

mult' = < ec', mem, objs-graphs' >

mult = < ec, mem, objs-graphs >

obj-graph = < pos-ec, taille-obj, nom-tamp, pos-tamp >

obj-graph' = < pos-ec', taille-obj, nom-tamp, pos-tamp >

colonne (pos-ec') =

si (colonne (pos) plus colonne (taille-obj))

supérieur colonne (taille-ec (ec))

alors colonne (taille-ec (ec)) moins colonne (taille-obj)

sinon colonne (pos)

ligne (pos-ec') =

si (ligne (pos) plus ligne (taille-obj))

supérieur ligne (taille-ec (ec))

alors ligne (taille-ec (ec) moins ligne (taille-obj)

sinon ligne (pos)

objs-graphs' = ajout (supelt (objs-graphs, obj-graph), obj-graph')

ec' = réafficher_zone (ec'', objs-graphs, pos-ec, taille-obj, mem)

où ec'' = afficher_zone_tampon (ec''', pos-ec', pos-tamp, taille-obj, nom-tamp, mem)

où ec''' = effacer_zone (ec'''', pos-ec', taille-obj)

où ec'''' = effacer_zone (ec, pos-ec, taille-obj)

Opération effacer zone :

ECRAN, POSITION-ECRAN, POSITION-ECRAN -> ECRAN

Partie lexique

But : effacer à l'écran une zone déterminée par la position de son coin supérieur gauche (csg) à l'écran et sa taille (taille-zone). On détermine en premier lieu la position du coin inférieur droit de la zone à l'écran (cid). Ensuite, on supprime de la partie affichage de l'écran (affich) toute l'information se trouvant dans la zone déterminée par csg et cid.

Objets :

ec, ec' : **ECRAN**

csg, taille-zone, taille-ecr : **POSITION-ECRAN**

affich, affich' : T[**POSITION-ECRAN, INFORMATION**]

cid : **POSITION-ECRAN**

Partie formelle

Définition :

ec' = effacer_zone (ec, csg, taille-zone)

avec

ec = < taille-ecr, affich >

ec' = < taille-ecr, affich' >

cid = add_pos (csg, taille-zone)

affich' =

iter pour i de ligne (csg) à ligne (cid)

iter pour j de colonne (csg) à colonne (cid)

si appt (affich, < j,i >)

alors suppt (affich, < j,i >)

Opération add_pos : **POSITION, POSITION -> POSITION**

Partie lexique

But : définir l'addition de deux positions

Objets :

pos1, pos2 : **POSITION**

pos-res : **POSITION**

col, lig : **ENTIER**

Partie formelle

Définition :

pos-res = add_pos (pos1, pos2)

avec

pos-res = < col, lig >

col = colonne (pos1) plus colonne (pos2)

lig = ligne (pos1) plus ligne (pos2)

Opération sous_pos : **POSITION, POSITION -> POSITION**

Partie lexique

But : définir la soustraction de deux positions

Objets :

pos1, pos2 : **POSITION**
pos-res : **POSITION**
col, lig : **ENTIER**

Partie formelle

Définition :

pos-res = sous_pos (pos1, pos2)
avec
pos-res = < col, lig >
col = colonne (pos1) moins colonne (pos2)
lig = ligne (pos1) moins ligne (pos2)

Opération afficher_zone_tampon :
ECRAN, POSITION-ECRAN, POSITION, POSITION, NOM-TAMPON, MEMOIRE -> ECRAN

Partie lexique

But : afficher à l'écran (ec) à une certaine position (pos-ec) une partie de tampon déterminée par la position de son coin supérieur gauche (csg) dans le tampon et sa taille (taille-tamp).

On détermine en premier lieu la position du coin inférieur droit (cid-tamp) de la partie du tampon (tamp) qu'il faut afficher à l'écran. Ensuite, on insére (affiche à l'écran) dans la partie affichage de l'écran (affich), la partie de tampon déterminée par csg-tamp et cid-tamp de manière à ce que la position de son coin supérieur gauche (csg-tamp) coïncide avec la position pos-ec.

Objets :

ec, ec' : **ECRAN**
csg-tamp, taille-tamp, cid-tamp : **POSITION**
pos-ec, taille-zone, taille-ec : **POSITION-ECRAN**
nom-tamp : **NOM-TAMPON**

mem : **MEMOIRE**
tamp : **TAMPON**
affich, affich' : T[**POSITION-ECRAN, INFORMATION**]
i, j : **ENTIER**

Partie formelle

Définition :

ec' = afficher_zone_tampon (ec, pos-ec, csg-tamp, taille-tamp,
nom-tamp, mem)

avec

ec = < taille-ec, affich >
ec' = < taille-ec, affich' >
cid-tamp = add_pos (csg-tamp, taille-tamp)
tamp = acct (mem, nom-tamp)
affich' =
iter pour i de ligne (csg-tamp) à ligne (cid-tamp)
iter pour j de colonne (csg-tamp) à colonne (cid-tamp)
si appt (contenu(tamp), < j,i >)
alors insert (affich, pos-ec, acct (contenu(tamp), < j,i >))
colonne (pos-ec) plus 1
ligne (pos-ec) plus 1

Remarque : L'écran et les tampons sont créés vides. La non-appartenance d'un indice à une ce ces tables signifie que

- dans le cas d'un écran aucune information n'est affichée à cette position,
- dans le cas d'un tampon aucun information n'est présente à cette position dans le tampon.

Opération réafficher_zone :

ECRAN, OBJS-GRAPHS, POSITION-ECRAN, POSITION-ECRAN, MEMOIRE
-> ECRAN

Partie lexique

But : réafficher à l'écran (ec), dans une zone déterminée par la position de son coin supérieur gauche à l'écran (csg-zone) et sa taille (taille-zone), le contenu des objets graphiques dans l'ordre de la liste des objets graphiques. Il faut qu'un objet graphique qui précède un autre objet graphique dans la liste n'ait pas son contenu caché par le contenu de ce dernier.

On détermine en premier lieu la position du coin inférieur droit de la zone à l'écran (cid-zone). Ensuite, on parcourt la zone à réafficher ; pour chaque position de cette zone, on détermine si cette position appartient à un objet graphique (obj_graph_contenant_point).

L'opération obj_graph_contenant_point prend dans la liste des objets graphiques (objs-graphs) le premier objet graphique qui contient la position de la zone. Si un tel objet existe, on affiche à l'écran, à cette position dans la zone, l'information contenue à la position correspondante dans le tampon de l'objet graphique (afficher_écran_info_tampon).

Objets :

ec, ec' : **ECRAN**
csg-zone, taille-zone, taille-ec, cid-zone : **POSITION-ECRAN**
objs-graphs : **OBJS-GRAPHS**
mem : **MEMOIRE**
affich, affich' : T[**POSITION-ECRAN, INFORMATION**]
i, j : **ENTIER**

Partie formelle

Définition :

ec' = réafficher_zone (ec, objs-graphs, csg-zone, taille-zone, mem)
avec
ec = < taille-ec, affich >
ec' = < taille-ec, affich' >
cid-zone = add_pos (csg-zone, taille-zone)
ec' = ec
iter pour i de ligne (csg-zone) à ligne (cid-zone)
iter pour j de colonne (csg-zone) à colonne (cid-zone)
sobj = obj_graph_contenant_point (j, i, objs-graphs)
si non eq-s (sobj, svide)
alors afficher_écran_info_tampon (j, i, tête (sobj), ec, mem)

Opération obj_graph_contenant_point :

ENTIER, ENTIER, OBJS-GRAPHS -> S[OBJ-GRAPH]

Partie lexique

But : obtenir dans la liste des objets graphiques (objs-graphs) le premier objet graphique (s'il existe) qui contient le point (coord1, coord2). Si l'objet graphique n'existe pas l'opération renvoie une liste vide.

Objets :

coord1, coord2 : **ENTIER**
objs-graphs : **OBJS-GRAPHS**
sobj : S[**OBJ-GRAPH**]

Partie formelle

Définition :

sobj = obj_graph_contenant_point (coord1, coord2, objs-graphs)
avec
sobj =
iter pour obj-graph dans objs-graphs tant que eq-s (sobj, svide)
si appartient_point_obj_graph (obj-graph, coord1, coord2)
alors ajout (sobj, obj-graph)

Opération appartient_point_obj_graph :
OBJ-GRAPH, ENTIER, ENTIER -> BOOL

Partie lexique

But : détermine si le point (coord1, coord2) est situé géographiquement dans l'objet graphique obj-graph. On détermine d'abord la position du coin inférieur droit de l'objet à l'écran (cid-obj). Ensuite, on détermine si le point (coord1, coord2) se trouve dans l'objet graphique qui occupe une zone à l'écran déterminée par pos-écran (obj-graph) (position du coin supérieur gauche à l'écran de la zone) et cid-obj (position du coin inférieur droit à l'écran de la zone).

Objets :

coord1, coord2 : **ENTIER**
obj-graph : **OBJ-GRAPH**
booléen : **BOOL**

Partie formelle

Définition :

booléen = appartient_point_obj_graph (obj-graph, coord1, coord2)
avec
cid-obj = add_pos (pos-écran (obj-graph), taille-objet (obj-graph))

booléen =

(((ligne (pos-écran (obj-graph)) infeg coord2)
et (ligne (cid-obj) supeg coord2)))
et
(((colonne (pos-écran (obj-graph)) infeg coord1)
et (colonne (cid-obj) supeg coord1)))

Opération afficher_écran_info_tampon :
ENTIER, ENTIER, OBJ-GRAPH, ECRAN, MEMOIRE -> ECRAN

Partie lexique

But : afficher à la position (coord1, coord2) de l'écran ec, l'information qui doit être affichée à cette position dans l'objet graphique obj-graph. On détermine d'abord la position dans le tampon (post-effective) de l'information qui doit être affichée dans l'objet graphique à la position (coord1, coord2). Si le tampon tampon contient de l'information à la position post-effective, alors cette information est affichée à l'écran à la position (coord1, coord2).

Objets :

ec, ec' : **ECRAN**
coord1, coord2 : **ENTIER**
obj-graph : **OBJ-GRAPH**
tampon : **TAMPON**
mem : **MEMOIRE**

Partie formelle

Définition :

ec' = afficher_écran_info_tampon (coord1, coord2, objs-graphs, ec, mem)
avec
tampon = acct (mem, nom-tampon (obj-graph))
post-effective = add_pos (pos-tampon, sous_pos (< coord1, coord2 >,
pos-écran (obj-graph))
ec' =
si appt (contenu(tampon), post-effective)
alors insert (affichage (ec), < coord1, coord2 >,
acct (contenu (tampon), post-effective))

Opération MONTER :
MULTIFENETRAGE, OBJ-GRAPH -> MULTIFENETRAGE

Partie lexique

But : faire apparaître l'objet graphique obj-graph à l'écran c'est-à-dire la rendre entièrement visible. L'objet graphique obj-graph est placé au sommet de la pile des objets graphiques. Pour rendre l'objet graphique obj-graph entièrement visible, il faut :

- déplacer l'objet graphique (obj-graph) en tête de la liste objs-graphs,
- effacer la zone d'écran (appelée par la suite zone A) qu'occupe l'objet graphique (effacer-zone),
- afficher l'information qui est contenue dans l'objet graphique dans la zone A (afficher_zone_tampon).

Objets :

mult, mult' : **MULTIFENETRAGE**
ec, ec', ec" : **ECRAN**
obj-graph : **OBJ-GRAPH**
objs-graphs, objs-graphs' : **OBJS-GRAPHS**
mem : **MEMOIRE**

Partie formelle

Définition :

mult' = MONTER (mult, obj-graph)
avec
mult' = < ec', mem, objs-graphs' >
mult = < ec, mem, objs-graphs >
objs-graphs' = ajout (supelt (objs-graphs, obj-graph), obj-graph)

ec' = afficher_zone_tampon (ec", pos-écran (obj-graph),
pos-tampon (obj-graph),
taille-objet (obj-graph),
nom-tampon (obj-graph), mem)

où ec" = effacer_zone (ec, pos-écran (obj-graph),
taille-objet (obj-graph))

Opération DESCENDRE :
MULTIFENETRAGE, OBJ-GRAPH -> MULTIFENETRAGE

Partie lexique

But : placer l'objet graphique obj-graph au bas de la pile des objets graphiques. Pour placer l'objet graphique obj-graph au bas de la pile, il faut :

- le déplacer en fin de la liste des objets graphiques objs-graphs,
- effacer la zone d'écran (appelée par la suite zone A) qu'il occupe (effacer-zone),
- réafficher dans la zone A le contenu des objets graphiques rendus visibles suite au déplacement de l'objet graphique obj-graph au bas de la pile des objets graphiques (réafficher-zone).

Objets :

mult, mult' : **MULTIFENETRAGE**
ec, ec', ec" : **ECRAN**
obj-graph : **OBJ-GRAPH**
objs-graphs, objs-graphs' : **OBJS-GRAPHS**
mem : **MEMOIRE**

Partie formelle

Définition :

mult' = DESCENDRE (mult, obj-graph)
avec
mult' = < ec', mem, objs-graphs' >
mult = < ec, mem, objs-graphs >
objs-graphs' = ajoutrg (supelt (objs-graphs, obj-graph),
taille (objs-graphs) plus 1, obj-graph)
ec' = réafficher_zone (ec", objs-graphs', pos-écran (obj-graph),
taille-objet (obj-graph), mem)
où ec" = effacer_zone (ec, pos-écran (obj-graph),
taille-objet (obj-graph))

Opération DETRUIRE :
MULTIFENETRAGE, OBJ-GRAPH -> MULTIFENETRAGE

Partie lexique

But : détruire l'objet graphique obj-graph. Pour détruire l'objet obj-graph à l'écran, il faut :

- le supprimer de la liste des objets graphiques objs-graphs,
- effacer la zone d'écran (appelée par la suite zone A) qu'il occupe (effacer-zone),

- réafficher dans la zone A le contenu des objets graphiques rendus visibles suite à la destruction de l'objet graphique (réafficher_zone).

Objets :

mult, mult' : **MULTIFENETRAGE**

ec, ec', ec" : **ECRAN**

obj-graph : **OBJ-GRAPH**

objs-graphs, objs-graphs' : **OBJS-GRAPHS**

mem : **MEMOIRE**

Partie formelle

Définition :

mult' = DETRUIRE (mult, obj-graph)

avec

mult' = < ec', mem, objs-graphs' >

mult = < ec, mem, objs-graphs >

objs-graphs' = supelt (objs-graphs, obj-graph)

ec' = réafficher_zone (ec", objs-graphs', pos-écran (obj-graph),
taille-objet (obj-graph), mem)

où ec" = effacer_zone (ec, pos-écran (obj-graph),
taille-objet (obj-graph))

Opération SELECTION :

FILE-APPLICATION, OBJ-GRAPH, POSITION-ECRAN, MEMOIRE
-> FILE-APPLICATION

Partie lexique

But : sélectionner un composant, une entité d'un objet graphique, (obj-graph) comme par exemple un item d'un menu, une information dans une fenêtre, à la position pos de l'écran ec. Si une entité existe à la position pos de l'écran (déterminer_sélection) alors cette entité et l'identificateur de l'objet graphique contenant cette entité, sont placés dans la file de l'application file.

Objets :

file, file' : **FILE-APPLICATION**

obj-graph : **OBJ-GRAPH**

pos : **POSITION-ECRAN**
info : S[**INFORMATION**]
mem : **MEMOIRE**

Partie formelle

Définition :

file' = SELECTION (file, obj-graph, pos)
avec
info = déterminer_sélection (mem, obj-graph, pos)
file'=
si non eq-s (info, svide)
alors ajoutrg (file, taille (file) plus 1, < id-objet (obj-graph),
info >)
sinon file

Opération déterminer sélection :

MEMOIRE, OBJ-GRAPH, POSITION-ECRAN -> S[INFORMATION]

Partie lexique

But : obtenir l'entité sélectionnée par l'utilisateur dans l'objet graphique obj-graph à la position pos de l'écran. On détermine en premier lieu la position dans le tampon de l'information affichée à la position pos à l'écran (pos-eff). Si pos-eff appartient au tampon alors

- on recule dans le tampon pour obtenir le début de l'entité sélectionnée,
- on copie l'entité dans information.

Objets :

obj-graph : **OBJ-GRAPH**
pos : **POSITION-ECRAN**
information : S[**INFORMATION**]
mem : **MEMOIRE**
tampon : **TAMPON**

Partie formelle

Définition :

information = déterminer_sélection (mem, obj-graph, pos)

avec

tampon = acct (mem, nom-tampon (obj-graph))

pos-eff = add_pos (pos-tampon(obj-graph), sous_pos (pos, pos-écran (obj-graph)))

information =

si appt (tampon, pos-eff)

alors

iter tant que appt (tampon, pos-eff)

colonne (pos-eff) moins 1

colonne (pos-eff) plus 1

iter tant que appt (tampon, pos-eff)

ajoutrg (information, taille (information) plus 1, acct (tampon, pos-eff))

colonne (pos-eff) plus 1

Les objets graphiques interactifs sont de type **OBJ-GRAPH**. Cependant sur certains objets interactifs l'utilisateur peut effectuer des opérations supplémentaires. C'est le cas pour l'objet interactif fenêtre de type **FENETRE** qui hérite donc des opérations possibles sur **OBJ-GRAPH** et auxquelles d'autres opérations sont ajoutées.

TYPE **FENETRE**

Partie lexique

But : une fenêtre est un écran virtuel

Opérations associées : {héritage des opérations possibles sur **OBJ-GRAPH,** plus} , PAGE_SUIVANTE, PAGE_PRECEDENTE,

DEBUT_TEXTE, FIN_TEXTE

Partie formelle

Structure : **OBJ-GRAPH**

Les opérations possibles pour l'utilisateur sur l'objet interactif **FENETRE** sont définies de la manière suivante.

Opération PAGE_SUIVANTE :
MULTIFENETRAGE, FENETRE -> MULTIFENETRAGE

Partie lexique

But : à chaque fenêtre est associé un tampon qui contient des informations. La taille de l'information contenue dans un tampon peut dans certains cas être supérieure à la taille de la fenêtre de sorte que toute l'information du tampon ne peut être affichée dans la fenêtre. L'opération page_suivante permet d'afficher la page suivante du tampon dans la fenêtre fen. Si la dernière page du tampon n'est pas déjà affichée dans la fenêtre alors :

- on calcule la nouvelle position pos-tamp' à partir de laquelle on affichera l'information du tampon,
- on efface le contenu de la fenêtre (effacer_zone),
- on affiche la partie du contenu du tampon attachée à la fenêtre à partir de la position pos-tamp' (afficher_zone_tampon).

Objets :

mult, mult' : **MULTIFENETRAGE**
ec, ec', ec" : **ECRAN**
fen : **FENETRE**
objs-graphs, objs-graphs' : **OBJS-GRAPHS**
mem : **MEMOIRE**
pos-ecr, taille-obj : **POSITION-ECRAN**
pos-tamp, pos-tamp' : **POSITION**
nom-tamp : **NOM-TAMPON**
tampon : **TAMPON**

Partie formelle

Définition :

mult' = PAGE_SUIVANTE (mult, fen)
avec
mult' = < ec', mem, objs-graphs' >
mult = < ec, mem, objs-graphs >
fen = < pos-ecr, taille-obj, nom-tamp, pos-tamp >
fen' = < pos-ecr, taille-obj, nom-tamp, pos-tamp' >
tampon = acct (mem, nom-tamp)

ec' =

si (ligne (pos-tamp) plus ligne (taille-obj))

infeg ligne (taille-tampon (tampon))

alors

pos-tamp' = < colonne (pos-tamp),

ligne (pos-tamp) plus ligne (taille-obj) >

ec' = afficher_zone_tampon (ec", pos-ecr, pos-tamp',

taille-obj, nom-tamp, mem)

où ec" = effacer_zone (ec, pos-ecr, taille-obj)

Opération PAGE_PRECEDENTE :

MULTIFENETRAGE, FENETRE -> MULTIFENETRAGE

Partie lexique

But : cette opération permet d'afficher la page précédente du tampon dans la fenêtre fen. Si la première page du tampon n'est pas déjà affichée dans la fenêtre alors :

- on calcule la nouvelle position pos-tamp' à partir de laquelle on affichera l'information du tampon,
- on efface le contenu de la fenêtre (effacer_zone),
- on affiche la partie du contenu du tampon attachée à la fenêtre à partir de la position pos-tamp' (afficher_zone_tampon).

Objets :

mult, mult' : **MULTIFENETRAGE**

ec, ec', ec" : **ECRAN**

objs-graphs, objs-graphs' : **OBJS-GRAPHS**

fen : **FENETRE**

mem : **MEMOIRE**

pos-ecr, taille-obj : **POSITION-ECRAN**

pos-tamp, pos-tamp' : **POSITION**

nom-tamp : **NOM-TAMPON**

tampon : **TAMPON**

Partie formelle

Définition :

mult' = PAGE_PRECEDENTE (mult, fen)

avec

```
mult' = < ec', mem, objs-graphs' >
mult = < ec, mem, objs-graphs >
fen = < pos-ecr, taille-obj, nom-tamp, pos-tamp >
fen' = < pos-ecr, taille-obj, nom-tamp, pos-tamp' >
tampon = acct (mem, nom-tamp)
ec' =
        si ligne (pos-tamp) supérieur 1
        alors
                ligne (pos-tamp) =
                        si (ligne (pos-tamp) moins ligne (taille-obj))
                                        inférieur 1
                        alors 1
                        sinon ligne (pos-tamp) moins ligne (taille-obj)

                pos-tamp' = < colonne (pos-tamp), ligne (pos-tamp) >
                ec' = afficher_zone_tampon (ec", pos-ecr, pos-tamp',
                                                taille-obj, nom-tamp, mem)
                        où ec" = effacer_zone (ec, pos-ecr, taille-obj)
```

Opération FIN_TEXTE :

MULTIFENETRAGE, FENETRE -> MULTIFENETRAGE

Partie lexique

But : l'opération fin_texte permet d'afficher la dernière page du tampon dans la fenêtre fen. Reste représente le nombre de lignes de la dernière page du tampon. Si la dernière page du tampon n'est pas déjà affichée dans la fenêtre alors :

- on calcule la nouvelle position pos_tamp' à partir de laquelle on affichera l'information du tampon,
- on efface le contenu de la fenêtre (effacer-zone),
- on affiche la partie du contenu du tampon attachée à la fenêtre à partir de la position pos-tamp' (afficher_zone_tampon).

Objets :

mult, mult' : **MULTIFENETRAGE**
ec, ec', ec" : **ECRAN**
objs-graphs, objs-graphs' : **OBJS-GRAPHS**
fen : **FENETRE**
mem : **MEMOIRE**

pos-ecr, taille-obj : **POSITION-ECRAN**
pos-tamp, pos-tamp' : **POSITION**
nom-tamp : **NOM-TAMPON**
tampon : **TAMPON**
reste : **ENTIER**

Partie formelle

Définition :

mult' = FIN_TEXTE (mult, fen)
avec
mult' = < ec', mem, objs-graphs' >
mult = < ec, mem, objs-graphs >
fen = < pos-ecr, taille-obj, nom-tamp, pos-tamp >
fen' = < pos-ecr, taille-obj, nom-tamp, pos-tamp' >
tampon = acct (mem, nom-tamp)
reste =
si eq-entier (mod ((ligne (taille-tampon (tampon))
div ligne (taille-obj)), 0)
alors ligne (taille-obj)
sinon mod ((ligne (taille-tampon (tampon)) div ligne (taille-obj))

ec' =
si ligne (pos-tamp) inférieur
(ligne (taille-tampon (tampon)) moins reste)
alors
pos-tamp' = < colonne (pos-tamp),
ligne (taille-tampon (tampon)) moins reste >
ec' = afficher_zone_tampon (ec'', pos-ecr, pos-tamp',
taille-obj, nom-tamp, mem)
où ec'' = effacer_zone (ec, pos-ecr, taille-obj)

Opération DEBUT_TEXTE :

MULTIFENETRAGE, FENETRE -> MULTIFENETRAGE

Partie lexique

But : l'opération début_texte permet d'afficher la première page du tampon dans la fenêtre fen.
Si la première page du tampon n'est pas déjà affichée dans la fenêtre alors :

- on calcule la nouvelle position pos-tamp' à partir de laquelle on affichera l'information du tampon,
- on efface le contenu de la fenêtre (effacer_zone),
- on affiche la partie du contenu du tampon attachée à la fenêtre à partir de la position pos-tamp' (afficher_zone_tampon).

Objets :

mult, mult' : **MULTIFENETRAGE**
ec, ec', ec" : **ECRAN**
objs-graphs, objs-graphs' : **OBJS-GRAPHS**
fen : **FENETRE**
mem : **MEMOIRE**
pos-ecr, taille-obj : **POSITION-ECRAN**
pos-tamp, pos-tamp' : **POSITION**
nom-tamp : **NOM-TAMPON**
tampon : **TAMPON**

Partie formelle

Définition :

mult' = DEBUT_TEXTE (mult, fen)
avec
mult' = < ec', mem, objs-graphs' >
mult = < ec, mem, objs-graphs >
fen = < pos-ecr, taille-obj, nom-tamp, pos-tamp >
fen' = < pos-ecr, taille-obj, nom-tamp, pos-tamp' >
tampon = acct (mem, nom-tamp)
ec' =
si ligne (pos-tamp) supérieur 1
alors
pos-tamp' = < colonne (pos-tamp), 1 >
ec' = afficher_zone_tampon (ec", pos-ecr, pos-tamp', taille-obj, nom-tamp, mem)
où ec" = effacer_zone (ec, pos-ecr, taille-obj)

En guise de récapitulatif, nous donnons à la figure 4.3.3 la forêt des opérations de la spécification MULTIFENETRAGE.

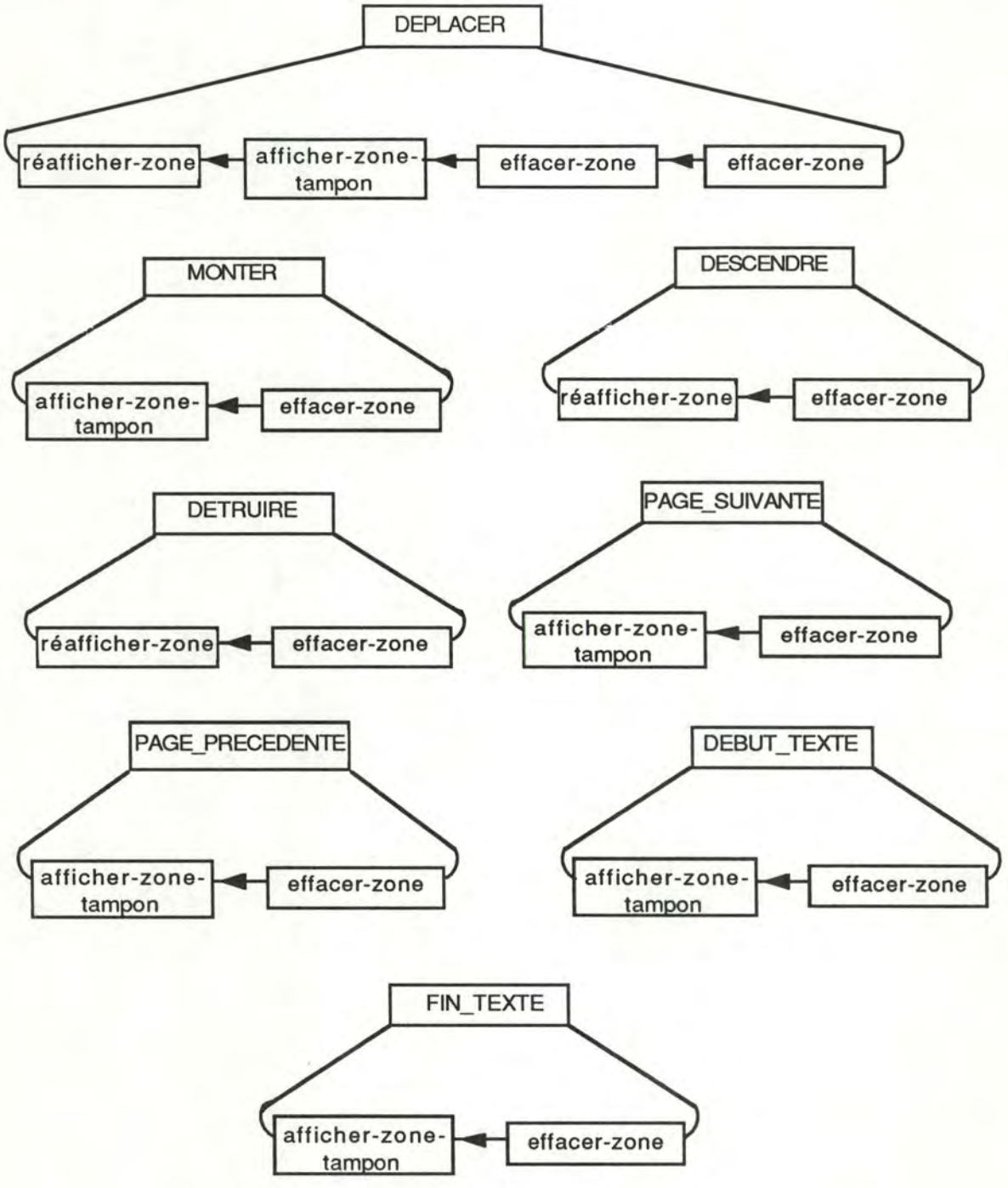


Figure 4.3.3 : forêt des opérations de la spécification multi-fenêtrage

La spécification en SACSO du gestionnaire de multi-fenêtrage décrit l'aspect "statique" de l'interface. En effet, les scénarios selon lesquels l'utilisateur va effectuer les différentes opérations ne sont pas précisés. Pour décrire cet aspect "dynamique" de l'interface, nous utilisons des **diagrammes de transition d'état** [Wasserman 82], [Wasserman 79].

Un diagramme de transition est un **réseau de noeuds** et d'**arcs orientés**. Chaque **noeud** est représenté par un cercle, qui représente un état stable, c'est-à-dire un état en dehors de toute transition d'état, attendant une commande (input) de l'utilisateur s'y appliquant. Chaque noeud a un nom unique, et un message de sortie peut être affiché lorsque le noeud est

atteint. Un noeud est désigné comme le noeud de départ et représenté par deux cercles concentriques.

Les noeuds sont reliés par des **arcs** qui représentent une transition d'état, c'est-à-dire le passage d'un état du système à un autre, basée sur une commande de l'utilisateur. Cette entrée est désignée par une chaîne de caractères comme par exemple 'charger', par un nom de variable, ou bien par le nom d'un autre diagramme entre < et > (exemple : < diagramme 2 >), de sorte que l'arc ne sera traversé que si le diagramme dont le nom est précisé sur l'arc est traversé. Lorsqu'un arc est laissé vide, il devient la transition par défaut c'est-à-dire que cet arc est emprunté lorsque la commande effective de l'utilisateur ne correspond à aucun des autres arcs.

Les **opérations** sont représentées par un rectangle contenant un entier. Une opération peut être associée à un arc. Cela signifie que cette opération sera exécutée lorsque l'arc sera traversé. Une même opération peut être associée à plusieurs arcs. Finalement dans un diagramme, un appel à un autre diagramme est représenté par un rectangle contenant le nom du diagramme appelé, délimité entre < et >.

Nous avons quelque peu modifié la syntaxe de Wasserman, pour permettre de décrire des interfaces utilisant des zones à sélection directe (exemple : menu, éditeur à boutons ...) :

- lorsque sur un arc figure le symbole !, cela signifie que la variable ou la chaîne de caractères qui suit le symbole ! n'est pas introduite au clavier mais par sélection directe comme par exemple en sélectionnant un item d'un menu,
- un arc sur lequel figure plusieurs arguments séparés par un blanc, ne sera traversé que si tous les arguments sont entrés par l'utilisateur.

A l'exception de ces modifications, la syntaxe de Wasserman a été conservée :

- une chaîne de caractères entre deux symboles ' sur un arc signifie que l'arc ne sera traversé que si l'utilisateur introduit cette chaîne de caractères au clavier suivi d'un retour chariot (return),
- des arcs sortant d'un rectangle représentant une opération peuvent contenir une valeur représentant une valeur renvoyée par l'opération et qui permet de prévoir un traitement différent suivant la valeur renvoyée par l'opération,
- le symbole + sur un arc signifie que cet arc est traversé sans que l'utilisateur ne doive faire une entrée.

L'interface de SACSO est présentée dans la suite au moyen d'un diagramme de transition d'état représentant les scénarios selon lesquels l'utilisateur peut lancer les différentes commandes dans le système. Ce diagramme fait appel à un autre diagramme de transition d'état. La signification des noeuds et opérations du premier diagramme de transition d'états présenté à la figure 4.3.4 et à la figure 4.3.5 (à placer en dessous de la figure 4.3.4) est la suivante :

noeud Start

Affichage de l'écran de niveau 1 de SACSO

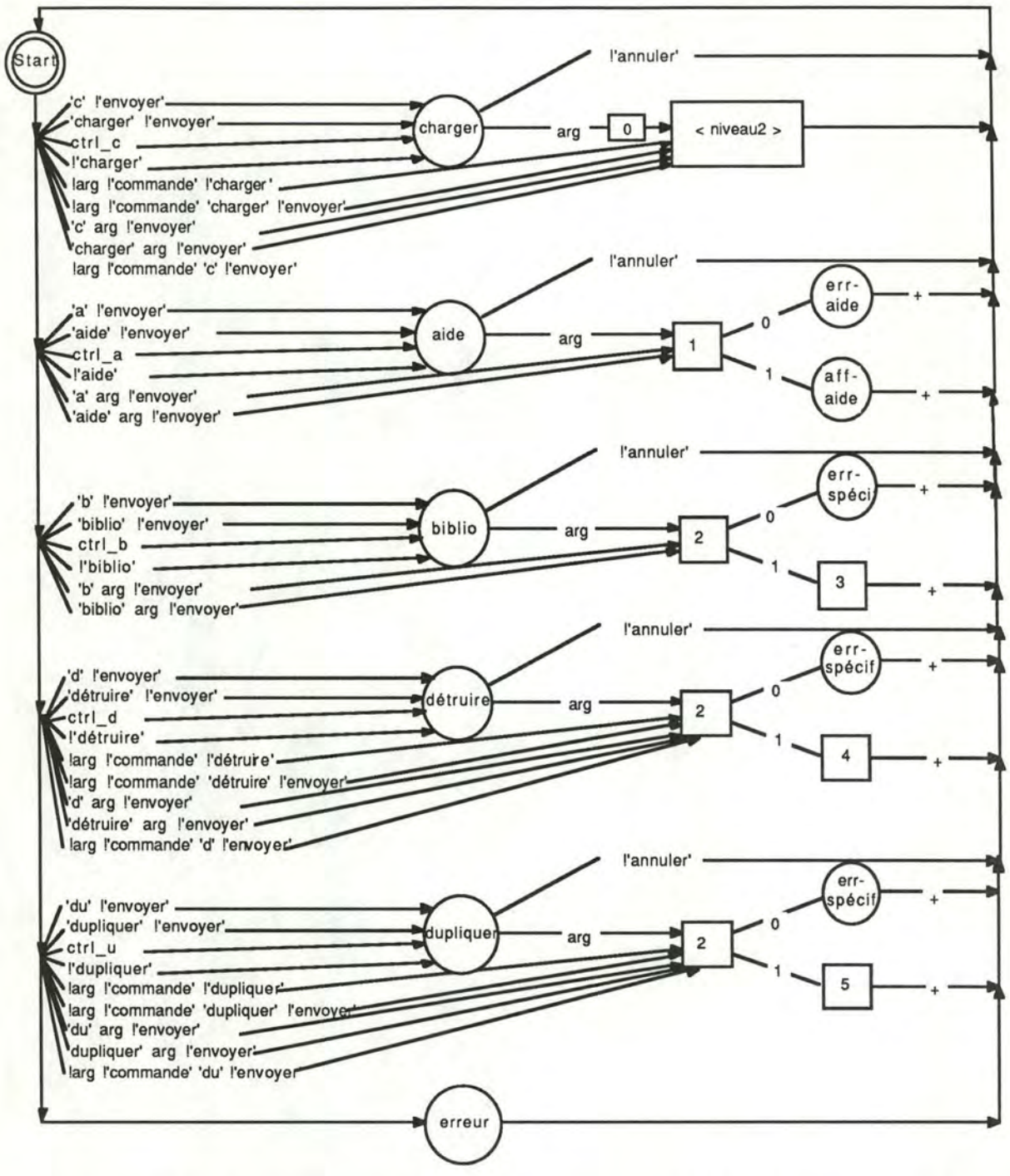


Figure 4.3.4 : diagramme de transition d'état du niveau 1 de SACSO (première partie)

noeud charger

"Veuillez introduire le nom de la spécification à charger"

opération 0

Chargement de la spécification dont le nom est contenue dans la variable arg

noeud aide

"Veuillez introduire le sujet sur lequel vous désirez de l'aide"

opération 1

Vérifier si arg correspond à un sujet pour lequel un texte d'aide peut être affiché
si oui renvoyer 1
si non renvoyer 0

noeud err-aide

"Désolé, il n'y a pas d'aide pour 'arg'"

noeud aff-aide

Affichage de l'aide correspondant au sujet arg

noeud biblio

"Veuillez introduire le nom de la spécification à mettre en bibliothèque"

opération 2

Vérifier l'existence d'une spécification de nom arg
si oui renvoyer 1
si non renvoyer 0

noeud err-spécif

"La spécification de nom 'arg' n'existe pas"

opération 3

Mise en bibliothèque de la spécification de nom arg

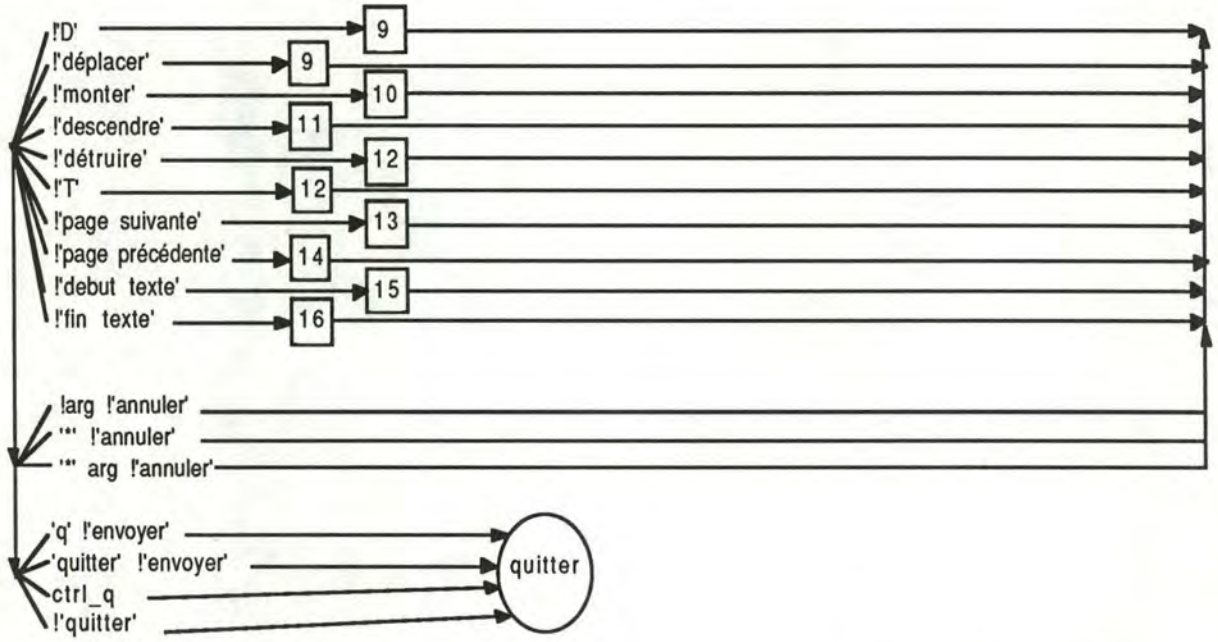


Figure 4.3.5 : diagramme de transition d'état du niveau 1 de SACSO (deuxième partie)

noeud détruire

"Veuillez introduire le nom de la spécification à détruire"

opération 4

Destruction de la spécification de nom arg

noeud dupliquer

"Veuillez introduire le nom de la spécification à dupliquer et son nouveau nom"

opération 5

Dupliquer la spécification de nom arg1 (première chaîne de caractères de arg) en lui donnant le nom arg2 (deuxième chaîne de caractères de arg)

< niveau2 >

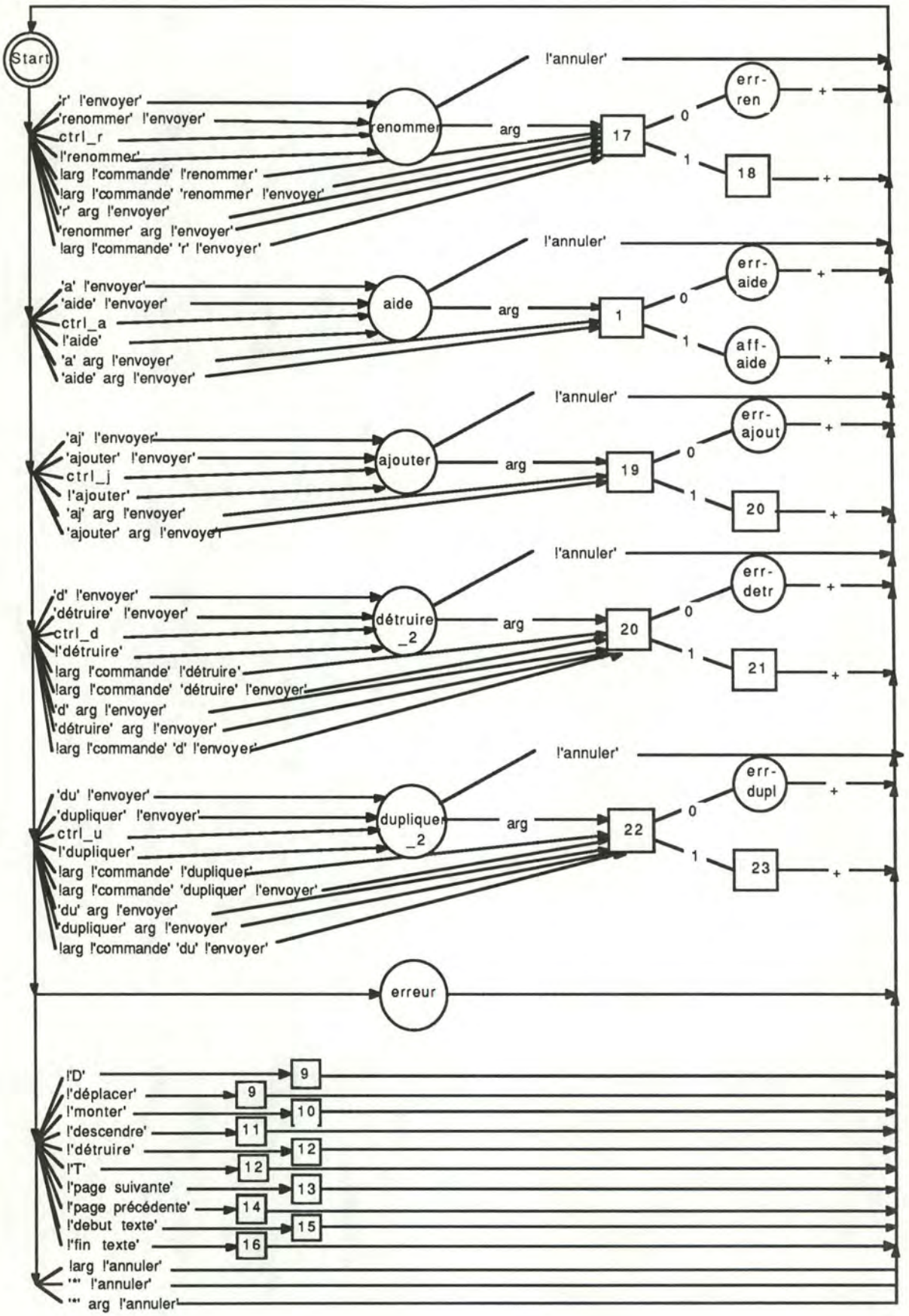


Figure 4.3.6 diagramme de transition du niveau 2 de SACSO (première partie)

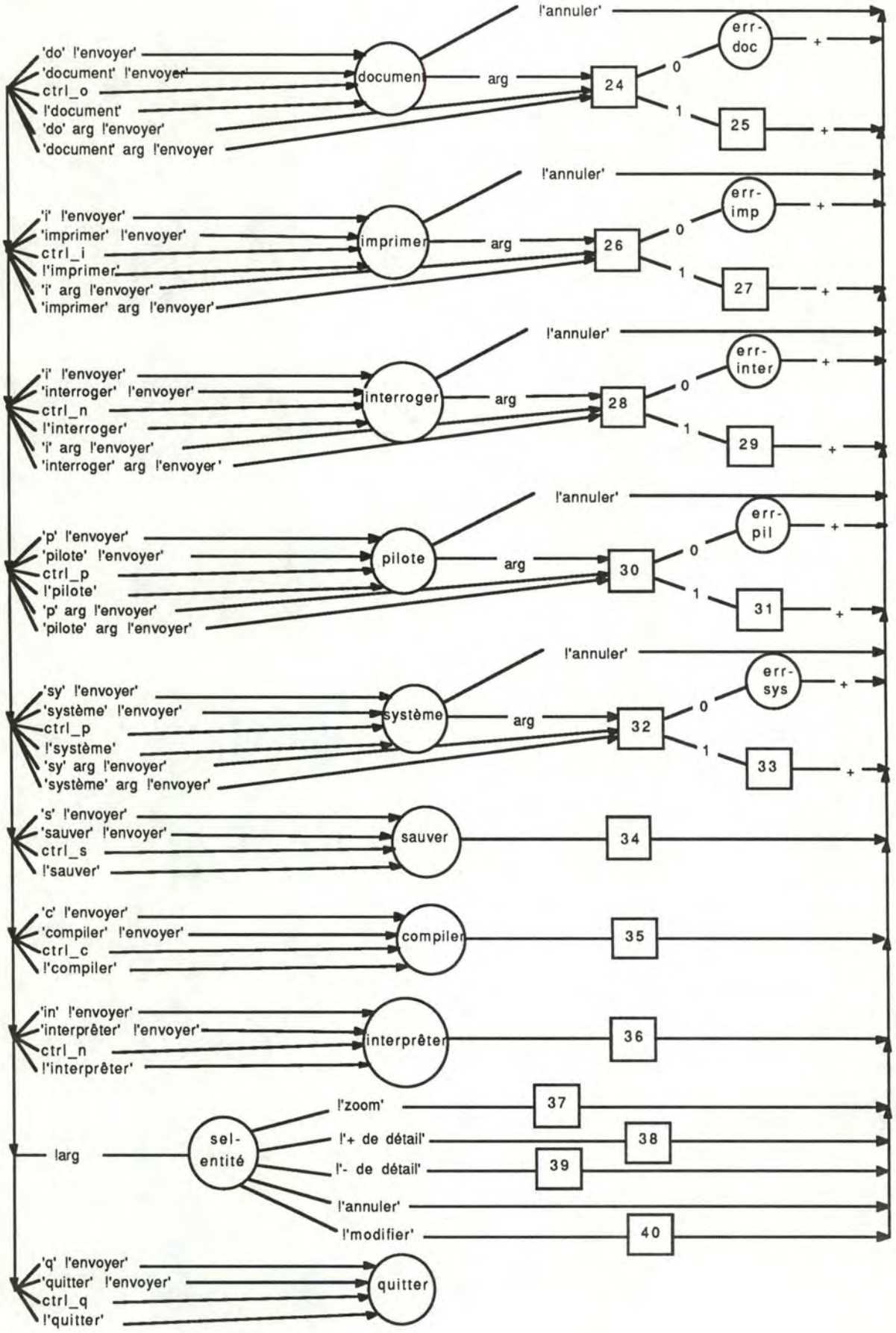


Figure 4.3.7 : diagramme de transition du niveau 2 de SACSO deuxième partie)

noeud erreur

"La commande introduite n'existe pas"

opération 9

Déplacer l'objet graphique sélectionné par l'utilisateur à la position indiquée par l'utilisateur

opération 10

Monter l'objet graphique désigné par l'utilisateur

opération 11

Descendre l'objet graphique désigné par l'utilisateur

opération 12

Détruire l'objet graphique désigné par l'utilisateur

opération 13

Afficher la page suivante dans l'objet graphique désigné par l'utilisateur

opération 14

Afficher la page précédente dans l'objet graphique désigné par l'utilisateur

opération 15

Afficher la première page dans l'objet graphique désigné par l'utilisateur

opération 16

Afficher la dernière page dans l'objet graphique désigné par l'utilisateur

noeud quitter

"Au revoir, à bientôt"

Pour la signification des noeuds et opérations du diagramme de transition du niveau 2 de SACSO présenté en figure 4.3.6 et 4.3.7 (à placer sous la figure 4.3.6) on consultera le manuel d'utilisation de SACSO.

4.3.3.3 Forme finale de la nouvelle interface de SACSO

Jusqu'à présent nous avons décrit les fonctionnalités du nouveau gestionnaire de multi-fenêtrage de SACSO et les scénarios selon lesquels l'utilisateur peut lancer ses commandes dans le système. Nous allons poursuivre notre démarche par niveau décroissant de "formel" en donnant de façon informelle la forme finale de l'interface de SACSO.

Chaque écran de la nouvelle interface de SACSO aura une présentation standardisée calquée sur les écrans décrits en section 1.7.2.

Les modifications apportées à l'interface sont les suivantes :

1) **Utilisation de la souris**

Le système de pointage avec curseur dirigé par des touches de contrôle (voir 1.7.2) peut avantageusement être remplacé par la souris. La souris permet à l'utilisateur d'aller pointer directement dans une fenêtre sur l'entité qu'il veut sélectionner et appuyer sur le bouton du milieu de la souris pour effectivement sélectionner l'entité. Lorsqu'il aura appuyé sur le bouton du milieu de la souris, s'affichera sur la console une boîte à messages (voir 2.12) indiquant l'entité qu'il a sélectionnée (partie MESSAGE de la boîte à messages) et lui présentant les opérations qu'il peut effectuer sur cette entité dans l'éditeur à boutons de la boîte à messages. Les opérations que l'utilisateur peut effectuer sur chaque entité n'ont évidemment pas été

changées (voir 1.7.2). Nous ajoutons simplement une opération ANNULER qui permet à l'utilisateur d'annuler l'effet du cliquage sur le bouton du milieu de la souris. L'utilisateur doit cliquer dans un des boutons de l'éditeur à boutons avec le bouton gauche de la souris. Deux cas sont dès lors possibles :

- l'utilisateur clique dans le bouton ANNULER de l'éditeur à boutons. Dans ce cas la boîte à messages est effacée et l'utilisateur se retrouve dans l'état avant le cliquage sur le bouton du milieu de la souris. (La nécessité d'un tel "**undo**" est soulignée par Schneiderman),
- l'utilisateur clique dans un autre bouton de l'éditeur à boutons. La boîte à messages est effacée (feedback) et l'opération attachée au bouton est exécutée.

Pour effectuer toutes ces manipulations, l'utilisateur n'utilisera que la souris.

La souris est également utilisée par l'utilisateur pour la gestion des fenêtres à l'écran c'est-à-dire pour déplacer les fenêtres, pour placer une fenêtre au sommet (bas) de la pile de fenêtres, pour détruire une fenêtre.

Pour remplir cette fonction de gestion des fenêtres nous avons ajouté à chaque fenêtre une barre de titre qui sert uniquement pour la gestion des fenêtres. Chaque fenêtre se présentera donc en première approximation de la manière présentée à la figure 4.3.8.

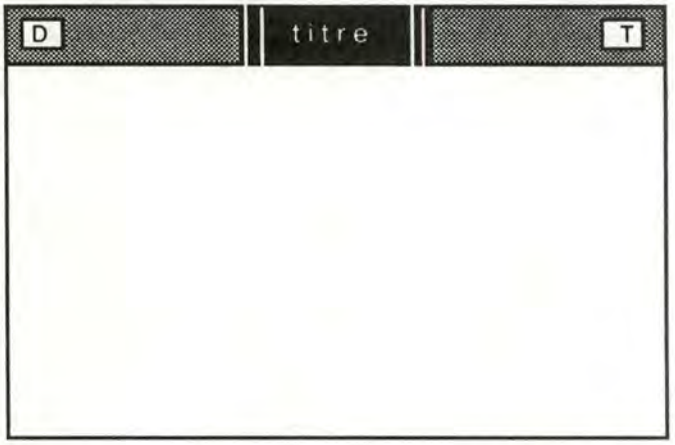


Figure 4.3.8 : présentation d'une fenêtre

En dessous du bord supérieur de la fenêtre se trouvera une barre de titre. Chaque zone de la barre de titre remplit une fonction précise pour la gestion des fenêtres :

- boîte à sélection gauche : permet de déplacer la fenêtre. Lorsque l'utilisateur voudra déplacer une fenêtre il lui suffira d'aller cliquer avec le bouton gauche de la souris dans cette boîte, de garder le bouton de gauche enfoncé, de déplacer la souris à l'endroit où il désire que le bord supérieur gauche de la fenêtre soit déplacé. En relâchant le bouton gauche de la souris, la fenêtre sera déplacée de manière telle que son bord supérieur gauche coïncide avec la position de la souris lors du relâchement du bouton gauche,

- boîte à sélection droite : permet de détruire la fenêtre. Lorsque l'utilisateur désire détruire la fenêtre il lui suffira de cliquer dans cette boîte avec le bouton gauche de la souris pour que la fenêtre soit détruite. Les fenêtres ne pouvant être détruites n'auront pas de boîte à sélection droite dans leur barre de titre,

- titre : permet de monter la fenêtre au sommet de la pile de fenêtres en cliquant avec le bouton gauche de la souris sur le titre de la barre de titre cette fenêtre,

- fond situé de part et d'autre du titre : permet de descendre la fenêtre au bas de la pile de fenêtres en cliquant avec le bouton gauche de la souris dans le fond de la barre de titre de cette fenêtre.

2) **Utilisation de menus à déroulement**

Le menu ne sera plus constitué par une fenêtre toujours visible à l'écran. Un menu à déroulement sera utilisé et sera affiché à l'écran lorsque l'utilisateur cliquera sur le bouton droit de la souris. Ce menu contiendra tous les libellés des commandes qui peuvent être exécutées au niveau où se trouve l'utilisateur. En accord avec les recommandations de Schneiderman, nous avons fait correspondre à chaque sélection du menu une touche de contrôle sur le clavier qui permet à des utilisateurs **experts** de lancer l'éxécution d'une commande à partir du clavier. D'autre part, toujours en suivant les principes de Schneiderman, nous avons limité à un niveau les sous-menus attachés à un menu (aucun menu n'est attaché à un sous-menu) ; on évite ainsi à l'utilisateur de passer par plusieurs sous-menus pour pouvoir trouver la sélection qu'il désire.

3) **Utilisation d'éditeurs radio pour gérer le défilement du texte dans la fenêtre**

Lorsque du texte est affiché dans une fenêtre, par exemple un message d'erreur, il se peut que le texte du message soit trop grand pour être affiché dans la fenêtre. Dès lors un mécanisme de défilement du texte dans la fenêtre est nécessaire. Ce mécanisme de défilement est contrôlé par un éditeur radio présent dans chaque fenêtre avec quatre boutons dont les libellés sont les suivants : page précédente, page suivante, début de texte, fin de texte. Lorsque l'utilisateur veut faire défiler le texte, il doit cliquer avec le bouton gauche de la souris dans un des boutons de l'éditeur radio. La présentation finale d'une fenêtre est illustrée à la figure 4.3.9.

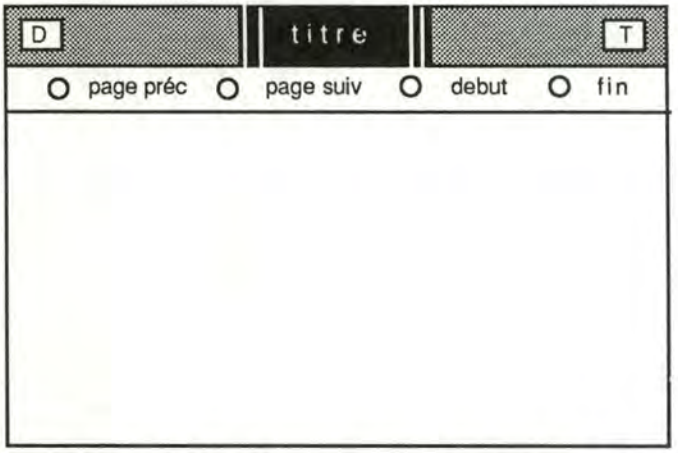


Figure 4.3.9 : présentation finale d'une fenêtre

4) **Utilisation du clavier**

Le clavier ne sera plus utilisé que pour entrer du texte dans l'éditeur de texte et, par les utilisateurs experts, pour lancer l'exécution d'une commande du menu grâce aux touches "control" du clavier.
L'éditeur de texte est une fenêtre constituée de quatre zones :

- une barre de titre, comme pour les fenêtres et pour les mêmes raisons, à savoir la gestion des fenêtres,

- un éditeur à boutons avec quatre commandes possibles :
  + annuler : lorsque l'utilisateur veut annuler la commande il doit cliquer avec le bouton gauche de la souris dans ce bouton,
  + envoyer : pour envoyer le texte qui a été introduit par l'utilisateur dans l'éditeur de texte, qu'il s'agisse d'une commande ou du texte d'une spécification,
  + aide : affiche les commandes de l'éditeur dans une fenêtre d'aide créée à cet effet sur demande de l'utilisateur lorsqu'il clique avec le bouton gauche de la souris dans ce bouton,
  + description paramètres : affiche la description des paramètres possibles pour une commande, sur demande de l'utilisateur lorsqu'il clique avec le bouton gauche de la souris dans ce bouton.

- une fenêtre éditeur de texte : il s'agit de la fenêtre qui contient le curseur. L'utilisateur dispose des commandes classiques d'édition, à savoir effacer un caractère, faire défiler le texte etc... Une particularité de cet éditeur est que l'utilisateur peut positionner le curseur à n'importe quel endroit de la fenêtre éditeur de texte grâce à la souris,

- zone message : cette zone est constituée d'une "fenêtre-texte" dans laquelle s'affichent les messages d'aide à l'utilisateur lorqu'il lance des commandes. Ces messages évitent à l'utilisateur de devoir mémoriser trop d'informations dans sa mémoire à court terme en lui rappelant au besoin ce qu'il doit faire. L'éditeur de texte est présenté à la figure 4.3.10.

Figure 4.3.10 : illustration de la fenêtre éditeur de texte

Nous pouvons maintenant passer à l'étape de conception du nouveau multi-fenêtrage pour SACSO.

4.3.4 Troisième étape : la conception

Cette section ne reprend que les aspects de la conception globale. Nous avons essayé, dans une phase de maintenance et d'extension du projet, de ne pas apporter trop de modifications au système. Il nous a paru nécessaire de localiser en premier lieu ces modifications et ensuite d'essayer de les concentrer afin de minimiser les répercussions sur le système existant.
Ne disposant pas d' une architecture logique, nous avons dû la construire à posteriori à partir du système existant pour faciliter l'étude et la localisation des modifications.
La démarche entreprise est à l'opposé de la démarche classique puisque nous avons dégagé cette architecture à partir du système déjà implémenté!
Nous présentons d'abord l'architecture, puis les modifications opérées.

4.3.4.1 Architecture logique du système

1) Définition

La démarche de conception d'une architecture logique consiste à structurer le système en composants et en relations bien définies entre ces composants (par exemple : exporte/importe, utilise, ...). Tout composant de l'architecture logique constitue une unité de spécification, de conception détaillée (ordre de conception, répartition entre programmeurs), de validation (jeux de tests, procédure de conduite de ces tests) et de modification. Tout composant est associé soit à un traitement (traitement dérivé d'une opération spécifiée lors de l'étape précédente), soit à une structure de données (type abstrait associé à un concept du problème).

L'architecture logique est structurée **hiérarchiquement** mettant en évidence des **niveaux** distincts et ordonnés suivant une **relation** bien définie entre les composants. Il existe autant de hiérarchies possibles que de relations possibles.
Dans cette architecture, le niveau 0 est défini comme étant l'ensemble des composants du système n'ayant aucune relation avec les composants de niveau inférieur. Le niveau i est défini comme étant l'ensemble des composants tels que si un des composants a une relation, c'est avec un composant de niveau inférieur à i.
Le critère choisi pour cette découpe est la relation **UTILISE** définie comme suit :

> "un composant A utilise un composant B lorsque le fonctionnement correct de A dépend de la disponibilité d'une version correcte de B."

2) Avantages

Dans une optique de transformation du système, cette découpe permet une compréhension au niveau logique indépendamment de toutes contraintes physiques d'implémentation. Elle permet de bien cerner l'importance ainsi que l'agencement des différents composants logiques et donne ainsi une facilité de maintenance du système.

3) Représentation

L'architecture logique est représentée par un graphe dont les noeuds représentent les différents composants logiques ou "modules" et les flèches représentent les relations entre les modules.
D'une manière générale une hiérarchisation UTILISE peut donner les niveaux suivants :

niveau 6 : fonctions dégagées lors de l'analyse fonctionnelle

niveau 5 : noyau fonctionnel : tri, insertion, extraction d'après critères, ...

niveau 4 : niveau entrées/sorties : gestionnaire de multi-fenêtrage, saisie de données, ...

niveau 3 : modules de contrôle : modules de contrôle dynamique (synchroniseur, séquenceur, ...), ...

niveau 2 : modules outils : pipes UNIX, gestionnaire d'écran, ...

niveau 1 : modules O.S. : UNIX, ...

Lors de l'analyse de SACSO, nous avons identifié la hiérarchie suivante dans laquelle les trois derniers niveaux de la démarche générale de hiérarchisation, ont été regroupés en un seul appelé niveau modules O.S.

niveau 5 : fonctions dégagées lors de l'analyse fonctionnelle

niveau 4 : noyau fonctionnel : primitives de manipulation, contrôles, ...

niveau 3 : entrées/sorties d'une spécification : affichage d'une spécification, saisie d'une spécification, ...

niveau 2 : modules entrées/sorties : gestionnaire de multi-fenêtrage, contrôle lexical et syntaxique, ...

niveau 1 : modules O.S.

Dans les différents niveaux, nous avons identifié les modules suivants :

niveau 5 : modules fonctionnels

L'ensemble de ces modules travaillent sur une spécification. Création, duplication, mise en bibliothèque, modification etc., sont les opérations diponibles sur une spécification.

niveau 4 : noyau fonctionnel

module **PRIMITIVES DE MANIPULATION**

Ce module regroupe l'ensemble des primitives de manipulation de composants d'une spécification. Une spécification est composée de types, d'opérations, d'objets, et d'un environnement. Il est possible de créer, de modifier, de supprimer un type, une opération, ou un objet. Il est possible d'ajouter ou de supprimer une spécification de l'environnement.

module **CONTROLES**

Ce module regroupe l'ensemble des contrôles sémantiques ainsi que les contrôles sur les incomplétudes.

module **DEDUCTIONS**

Ce module est responsable des inférences de types, il regroupe les primitives permettant d'inférer le profil d'une fonction, la structure d'un type, ou le type d'un objet à partir de ses utilisations.

niveau 3 : entrées/sorties d'une spécification

module **AFFICHAGE**

Ce module est responsable de l'affichage d'une spécification et de son réaffichage incrémental. Augmenter, ou diminuer le détail de l'affichage d'une spécification, faire un "zoom" de l'affichage d'une spécification, sont quelques fonctionnalités de ce module.

module **SAISIE**

Ce module est responsable de la désignation d'un noeud d'un arbre abstrait décrivant une spécification.

module **MANIPULATION D'ARBRES**

Il regroupe l'ensemble des primitives permettant de manipuler les structures d'arbre (arbre abstrait décrivant une spécification, structure intermédiaire décrivant la spécification affichée).

niveau 2 : modules entrées/sorties

module **MULTI-FENETRAGE**

Il regroupe l'ensemble des primitives de gestion des fenêtres, d'affichage d'informations dans ces fenêtres, ...

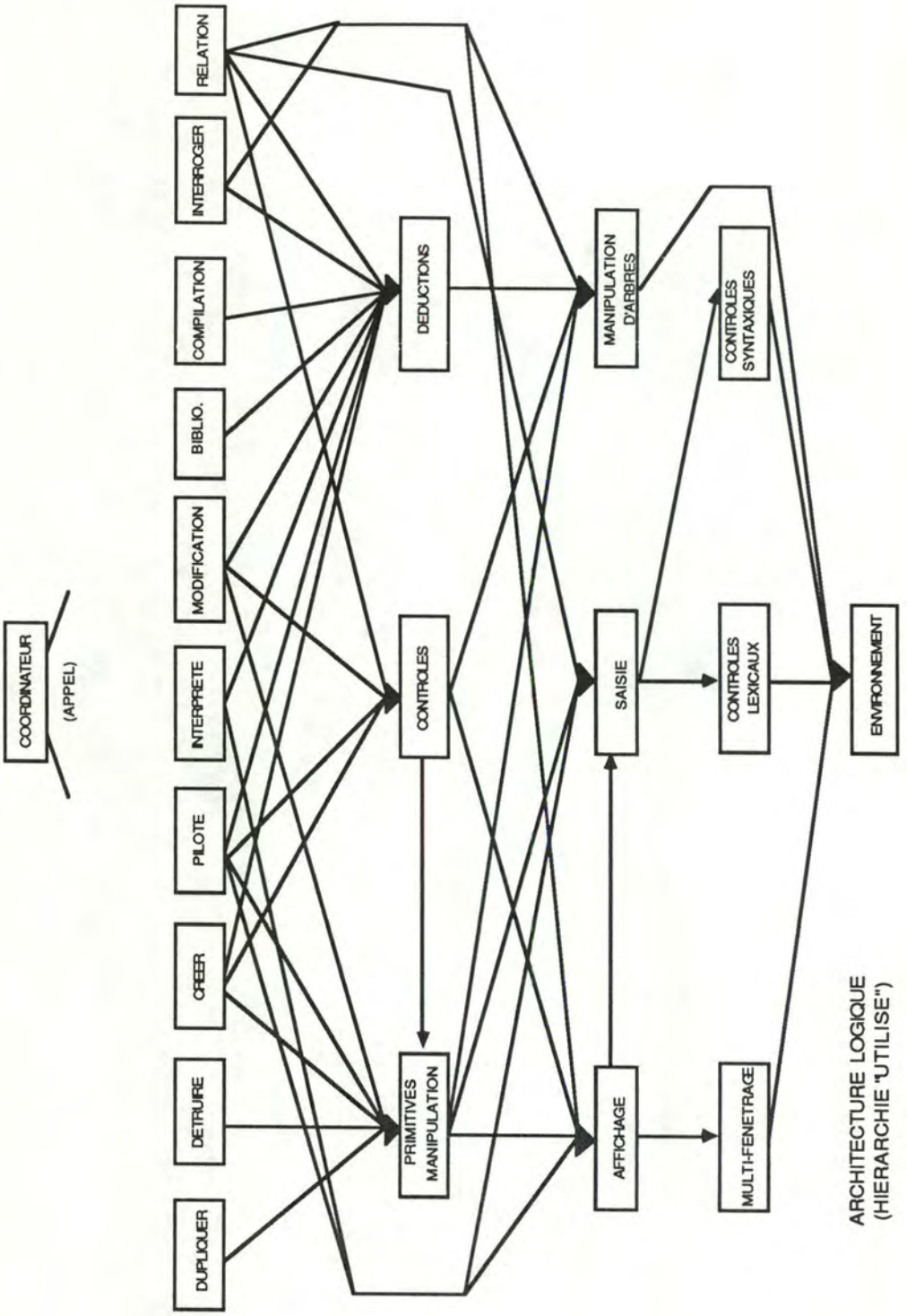


Figure 4.3.11 : l'architecture logique de SACSO

module **CONTROLES LEXICAUX**

Ce module regroupe l'ensemble des primitives qui définissent l'analyseur lexical chargé de l'analyse lexicale d'une phrase du langage SACSO.

module **CONTROLES SYNTAXIQUES**

Ce module regroupe l'ensemble des primitives qui définissent l'analyseur syntaxique chargé de l'analyse syntaxique d'une phrase du langage SACSO.

niveau 1 : environnement UNIX

L'architecture logique proposée est illustrée à la figure 4.3.11.

4) Remarque

Pour être complet dans notre démarche, nous devrions également spécifier complètement les différents modules logiques (conception détaillée). Vu le temps requis, nous nous sommes limités à la spécification des modules indispensables pour les modifications.

4.3.4.2 Localisation des modifications

Dans l'étape d'analyse des besoins, nous avons décidé entre autres, d'utiliser l'outil X Windows. Nous pensons également qu'il est nécessaire de modifier principalement les modules de gestion du multi-fenêtrage, d'affichage et de saisie. Nous introduisons dans notre hiérarchie deux niveaux supplémentaires entre le niveau modules O.S. et le niveau entrées/sorties d'une spécification. Ce qui nous donne la nouvelle architecture logique suivante :

niveau 7 : fonctions dégagées lors de l'analyse fonctionnelle

niveau 6 : noyau fonctionnel : primitives de manipulation, contrôles, ...

niveau 5 : entrées/sorties d'une spécification : affichage d'une spécification, saisie d'une spécification, ...

niveau 4 : modules entrées/sorties : gestionnaire de multi-fenêtrage, contrôle lexical et syntaxique, ...

niveau 3 : **boîte à outils étendue** : couche au-dessus de X Windows

niveau 2 : **outils de base** : X Windows

niveau 1 : modules O.S. (UNIX)

Le troisième niveau représente l'ensemble des primitives C utilisant les primitives X Windows du niveau inférieur et qui réalise **une couche au-dessus de X Windows**. Cette couche réalise en fait, une extension de X Windows. Son but est de fournir à une application, les moyens de construire un gestionnaire de multi-fenêtrage. Ce dernier remplacera le module de gestion de multi-fenêtrage de SACSO, utilisé par les primitives d'affichage et de saisie d'une spécification du niveau supérieur.

Les services offerts par cette couche s'étendent aux :

- notions graphiques de base (bitmap),
- notions textuelles (polices de caractères),
- notions liées au partage de l'écran (fenêtres, événements),
- notions d'entités simples (boutons, menus, etc.),
- notions d'entités composées (boîtes à messages).

Cette couche augmente le niveau d'abstraction des services offerts par X Windows (niveau 2). Pour construire le gestionnaire de multi-fenêtrage (niveau 4) utilisant les primitives de cette couche, le programmeur n'a plus à se soucier des structures de données des objets interactifs, à savoir les structures de "record" X Windows.
De plus, X Windows se présente comme un grand sac de fonctions en vrac. Le programmeur doit en extraire les fonctions et déterminer la colle d'assemblage. Par contre, la boîte à outils étendue (niveau 3) structure par objet interactif considéré, les différents traitements qui s'y rapportent et qui permettent la gestion de cet objet interactif.
Cependant, le niveau d'abstraction de cette couche est encore trop bas, au sens où l'implémenteur est tout de même contraint de chercher dans la boîte à outils X Windows les fonctions nécessaires à une extension éventuelle de la couche.
Dans la suite, nous présentons la découpe logique qui concerne la couche et le module de gestion du multi-fenêtrage.

4.3.4.3 Découpe logique de la couche : le niveau 3

Sachant que nous allons utiliser X Windows, nous allons en premier lieu inventorier les opérations qui devraient être fournies à un programmeur pour construire une interface ou un gestionnaire de multi-fenêtrage.
Il est important de noter que nous n'avons pas réalisé une couche objet (au sens de Smalltalk) puisqu'il n'est pas question ici d'héritage, de classe ou de transmissions de messages. Les objets sont représentés par des boîtes et les opérations disponibles pour chaque objet sont représentées autour de chaque objet. Une illustration est présentée aux figures 4.3.12, 4.3.13 et 4.3.14.

L'utilisateur grâce aux moyens d'entrée (voir section 2.9) tels que clavier et/ou souris, peut effectuer certaines opérations sur les objets identifiés précédemment. Il est évidemment possible lors de la construction d'une interface de restreindre ou d'étendre les actions que l'utilisateur peut réaliser sur chaque objet. Par exemple pour une fenêtre particulière, il est possible d'interdire sa destruction en inhibant la sélection DETRUIRE du menu qui sera attaché à cette fenêtre ou en supprimant la boîte à sélection droite qui sert à tuer la fenêtre. A titre d'exemple, nous présentons en caractère gras dans les figures 4.3.12, 4.3.13, et 4.3.14 les opérations qu'un utilisateur de SACSO pourrait réaliser sur les différents objets.

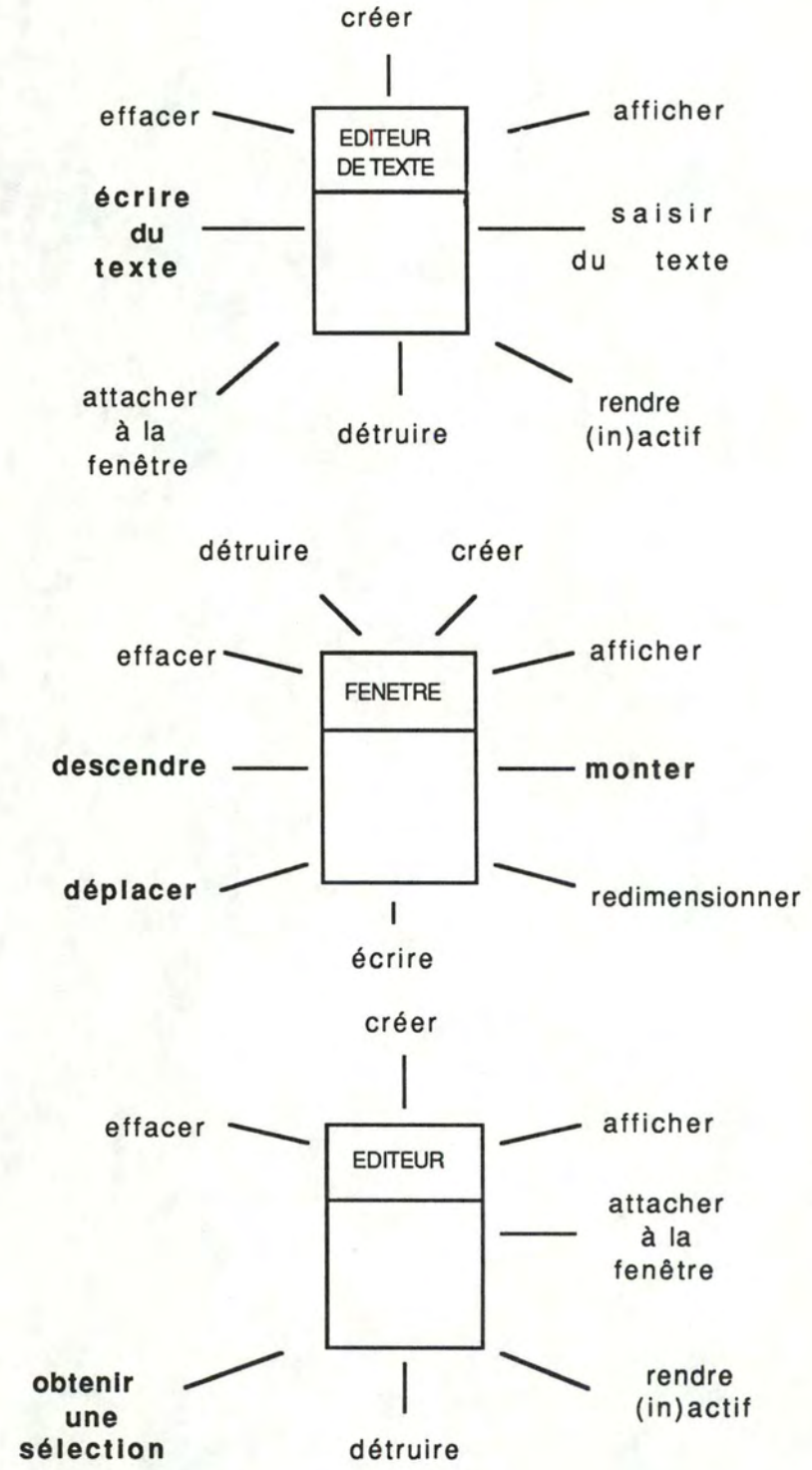


Figure 4.3.12 : objets disponibles et opérations nécessaires (première partie)

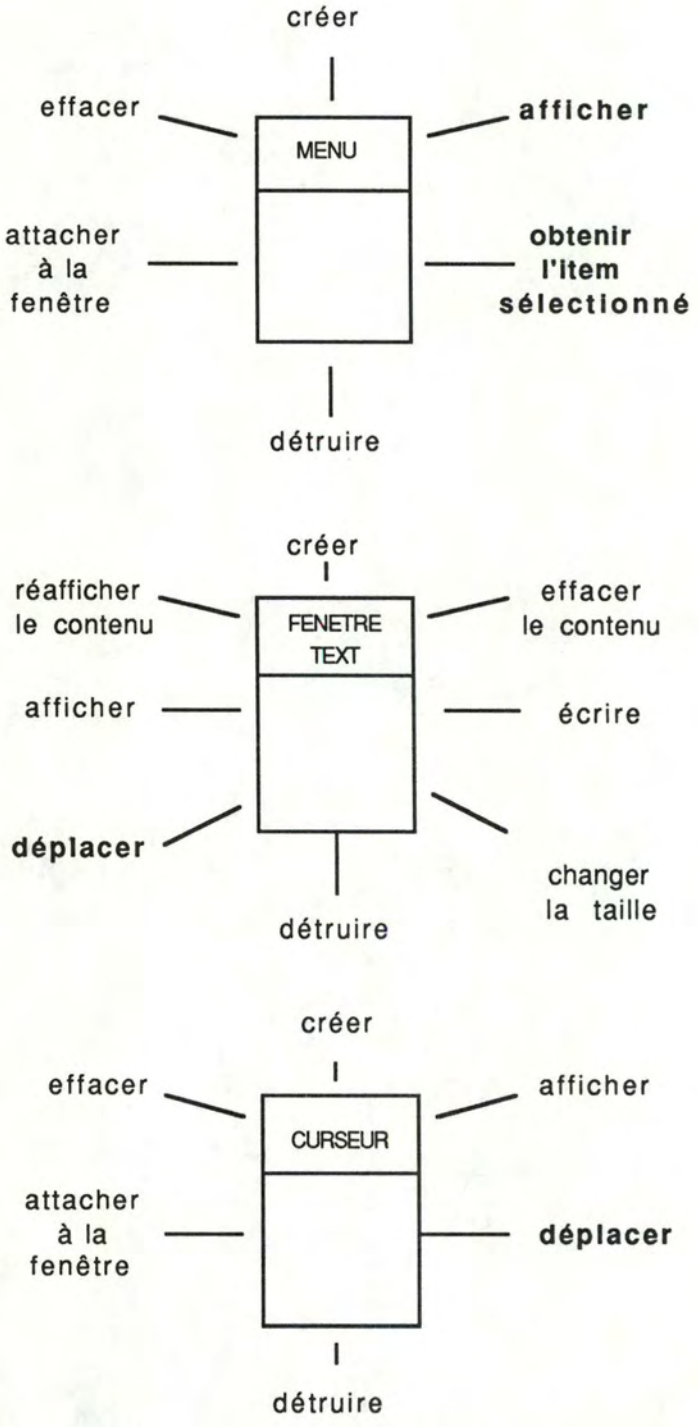


Figure 4.3.13 : objets disponibles et opérations nécessaires (deuxième partie)

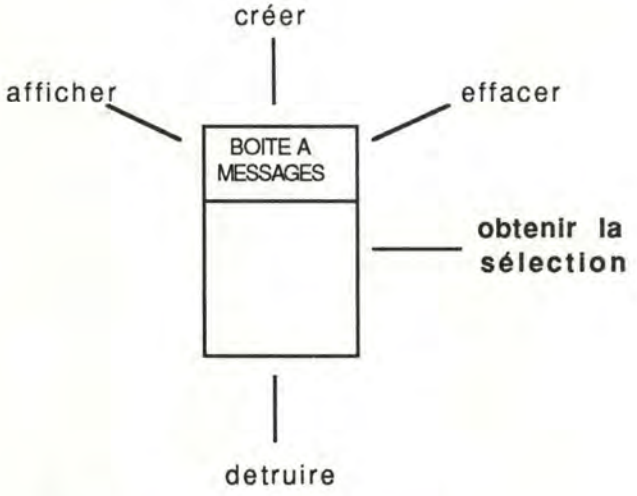


Figure 4.3.14 : objets disponibles et opérations nécessaires (troisième partie)

Nous regroupons sous le terme "EDITEUR" un certain nombre d'objets interactifs qui permettent à l'utilisateur de choisir une action parmi différentes alternatives proposées. Ce regroupement se justifie uniquement pour des raisons de similitude des opérations qu'il est possible d'effectuer sur ces différents objets interactifs.
Nous devons pouvoir créer, détruire, afficher, effacer une instance de n'importe quel objet.
Les objets de type "MENU", "EDITEUR", "BOITE A MESSAGES" permettent de présenter, suivant différentes formes de présentation, les actions qu'il est possible de réaliser. Il est donc nécessaire de pouvoir obtenir la sélection de l'utilisateur. D'où la fonction "obtenir une sélection" pour ces objets.
Certains objets comme ceux de type "MENU" ou de type "CURSEUR" doivent être liés à une fenêtre et ainsi permettre un traitement différent suivant la fenêtre dans laquelle se trouve l'utilisateur. Il faut donc prévoir, pour ces types d'objets, une fonction "attacher à la fenêtre".
Les objets comme ceux de type "EDITEUR" et "EDITEUR DE TEXTE", qui doivent également être liés à une fenêtre, peuvent être désactivés (respectivement activés) afin d'inhiber (respectivement permettre) les actions de l'utilisateur sur ces objets.
Les objets de type "FENETRE" doivent pouvoir être déplacés ou redimensionnés, tout en déplaçant et redimensionnant si nécessaire, les autres objets qui leurs sont spatialement liés.
Lorsqu'il est partiellement ou entièrement caché par un autre objet, un objet de type "FENETRE" doit pouvoir être placé au sommet de la pile de fenêtres. Dans le cas inverse, il doit pouvoir être placé au bas de la pile de fenêtres.
Un objet de type "FENETRE TEXT" est conçu pour y afficher uniquement du texte. Une gestion de buffer est prévue pour permettre un réaffichage du contenu en cas "d'exposition" (cfr 2.7). A l'opposé, le contenu de l'objet fenêtre n'est pas limité au texte et peut comprendre du graphisme. Cependant il n'est pas possible de réafficher son contenu, aucune gestion de buffer n'étant prévue.

4.3.4.4 Découpe logique du gestionnaire de multi-fenêtrage de niveau 4

Comme nous l'avons vu dans l'étape de spécification fonctionnelle (cfr. 4.3.3.2 et 4.3.3.3), la spécification de notre interface met en évidence un certain nombre de types d'objet (fenêtre, fenêtre éditeur de texte, menu, etc.) et un certain nombre d'opérations (monter, détruire, etc.) à effectuer sur les occurrences de ces types d'objet.
Considérons **ENS_TYPE_OBJET** l'ensemble des types d'objet suivant (cfr. 4.3.3.2 et 4.3.3.3) :

FENETRE, FENETRE_EDITEUR_DE_TEXTE, BOITE_A_MESSAGES, MENU.

Considérons également **ENS_ID_OPERATION** l'ensemble des identificateurs d'opération suivant (cfr. 4.3.3.2 et 4.3.3.3) :

DEPLACER, DETRUIRE, MONTER, DESCENDRE, DEBUT_TEXTE, FIN_TEXTE, PAGE_PRECEDENTE, PAGE_SUIVANTE, SELECTION, EDITION_DE_TEXTE, ENVOYER_COMMANDE, ANNULER_COMMANDE, AIDE_COMMANDE, DESC_PARAMETRES_COMMANDE.

Pour les objets de type FENETRE, l'ensemble **ENS_ID_OP_FENETRE** des identificateurs d'opération possibles, est le suivant (cfr.4.3.3.2) :

DEPLACER, DETRUIRE, MONTER, DESCENDRE, DEBUT_TEXTE, FIN_TEXTE, PAGE_PRECEDENTE, PAGE_SUIVANTE, SELECTION.

Pour les objets de type FENETRE_EDITEUR_TEXTE, l'ensemble **ENS_ID_OP_FENETRE_EDITEUR_TEXTE** des identificateurs d'opération possibles, est le suivant (cfr. 4.3.3.2 et 4.3.3.3) :

DEPLACER, MONTER, DESCENDRE, SELECTION, EDITION_DE_TEXTE, ENVOYER_COMMANDE, ANNULER_COMMANDE, AIDE_COMMANDE, DESC_PARAMETRES_COMMANDE.

Pour les objets de type MENU ou BOITE_A_MESSAGES, l'ensemble **ENS_ID_OP_MENU** ou **ENS_ID_OP_BOITE_A_MESSAGES** des identificateurs d'opération possibles, est le suivant (cfr. 4.3.3.2) :

SELECTION.

Ces opérations représentent les actions qu'un utilisateur peut effectuer sur les différents objets. Chaque action pourrait faire l'objet d'un seul module logique chargé de sa gestion. Cependant, chacun de ces modules aurait la même fonctionnalité qui est de gérer une action de l'utilisateur. C'est pourquoi l'ensemble de la gestion de ces actions est confiée à un seul module, le module de gestion des actions utilisateur.
D'autres modules logiques sont également nécessaires. Ils représentent ce qu'un gestionnaire de multi-fenêtrage doit offrir à un programmeur pour que celui-ci puisse créer une interface. Ces modules indispensables à un programmeur, sont la création et l'affichage d'objets ainsi que le module de gestion des entrées.
La découpe logique fait donc apparaître quatre modules responsables des opérations possibles sur les objets (figure 4.3.15).

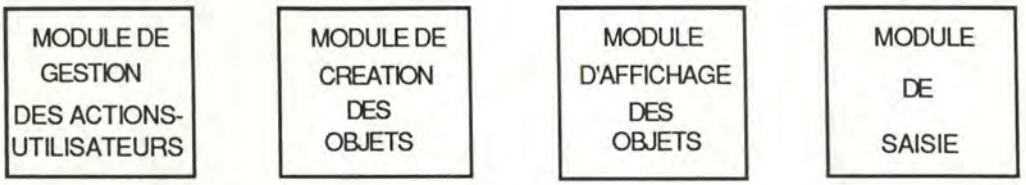


Figure 4.3.15 : découpe du gestionnaire de multi-fenêtrage (niveau 4)

Le module de saisie est responsable de tout ce qui concerne les entrées, à savoir la gestion du clavier et/ou de la souris. Il est également chargé d'analyser ces entrées pour déterminer l'action utilisateur correspondant à ce qui a été saisi. En ce sens, le module de saisie joue un peu le rôle d'un "coordinateur" ayant avec le module de gestion des actions utilisateur des relations de type "appel".

Spécifications des modules

1) Module de gestion des actions de l'utilisateur : GESTION-DES-ACTIONS-UTILISATEUR

a) But

Ce module est utilisé pour la gestion des opérations qu'un utilisateur peut effectuer sur les objets.
Il doit également tenir compte de la particularité des objets interactifs concernés et ne permettre que les opérations possibles compte tenu du type des objets.

b) Spécification externe

- *arguments* :

NOM_OBJET	:	nom de l'objet
TYPE_OBJET	:	type de l'objet
ID_OPERATION	:	identificateur de l'opération
ARGUMENTS	:	arguments de l'opération

- *pré-conditions* :

NOM_OBJET identifie un objet de type TYPE_OBJET.
TYPE_OBJET identifie le type d'un objet. Ce type est connu du système, il appartient à ENS_TYPE_OBJET.
ID_OPERATION identifie l'opération. Cet identificateur est connu du système, il appartient à ENS_ID_OPERATION.
ARGUMENTS est la liste des arguments supplémentaires nécessaires pour effectuer l'opération identifiée par ID_OPERATION. Ces arguments sont de types différents suivant l'identificateur d'opération. Pour les identificateurs d'opération suivants :

- MONTER, DESCENDRE, DETRUIRE : les arguments sont la pile des objets graphiques, et un écran,
- DEPLACER : les arguments sont une position écran, la pile des objets graphiques et un écran,
- SELECTION : les arguments sont une position, et la file de l'application contenant les commandes effectuées par l'utilisateur qui doivent être traitées par l'application,
- ENVOYER_COMMANDE (ANNULER, AIDE, DESC_PAR) : l'argument est le texte saisi.

- *résultat* :

L'opération identifiée par ID_OPERATION est appliquée à l'objet de nom NOM_OBJET, avec les arguments ARGUMENTS.

- *post-conditions* :

Si l'identificateur ID_OPERATION n'appartient pas à l'ensemble des identificateurs d'opération possibles sur l'objet de type TYPE_OBJET (par exemple, ENS_ID_OP_FENETRE pour les objets de type FENETRE, ou ENS_ID_OP_MENU pour les objets de type MENU),
alors l'exécution de ce module est sans effet,
sinon l'écran d'affichage est "modifié" en fonction de l'identificateur d'opération suivant :

- DEPLACER : cette opération a pour effet de déplacer l'objet de nom NOM_OBJET (et son contenu) à une autre position à l'écran. Déplacer un objet le fait apparaître entièrement à l'écran. Cet objet est placé au sommet de la pile des objets graphiques,
- MONTER (DESCENDRE) : cette opération a pour effet de placer l'objet de nom NOM_OBJET au sommet (bas) de la pile des objets graphiques,
- DETRUIRE : cette opération a pour effet de détruire l'objet de nom NOM_OBJET. Cet objet est enlevé de la pile des objets graphiques,
- SELECTION : cette opération n'a pas d'effet sur l'écran d'affichage. Elle ajoute le composant sélectionné dans l'objet de nom NOM_OBJET et l'identificateur de cet objet dans la file de l'application,
- PAGE_PRECEDENTE (SUIVANTE) : cette opération a pour effet d'afficher la page précédente (suivante) du tampon associé à l'objet de nom NOM_OBJET,
- DEBUT_TEXTE (FIN_TEXTE) : cette opération a pour effet d'afficher la première (dernière) page du tampon associé à l'objet de nom NOM_OBJET,
- EDITION_DE_TEXTE : cette opération a pour effet d'afficher le contenu du tampon d'édition associé à l'objet de nom NOM_OBJET,
- ENVOYER_COMMANDE : cette opération a pour effet d'effacer le contenu du tampon d'édition associé à l'objet de nom NOM_OBJET,
- ANNULER_COMMANDE : cette opération a pour effet d'afficher le contenu du tampon d'édition antérieur à l'envoi de la commande qui est annulée. Ce tampon est associé à l'objet de nom NOM_OBJET,

- AIDE_COMMANDE, DESC_PAR_COMMANDE : cette opération a pour effet d'afficher un objet de type FENETRE placé au sommet de la pile des objets graphiques. Cet objet contient un texte d'aide pour les commandes de l'éditeur (AIDE_COMMANDE) ou pour les commandes de SACSO (DESC_PAR_COMMANDE),

2) Module de création des objets : CREATION-DES-OBJETS

a) But

Ce module est chargé de la création des objets de différents types. Un objet vu comme un tout par l'utilisateur mais il peut être composé d'objets plus élémentaires. Ce module doit en tenir compte.

b) Spécification externe

- *arguments* :

NOM_OBJET : nom de l'objet
TYPE_OBJET : type de l'objet
ACTIONS_UTILISATEUR : ensemble d'identificateurs d'action utilisateur possibles sur cet objet

- *pré-conditions* :

NOM_OBJET identifie un objet qui n'existe pas déjà.
TYPE_OBJET identifie le type d'un objet. Ce type est connu du système, il appartient à ENS_TYPE_OBJET.
ACTIONS_UTILISATEUR définit l'ensemble des identificateurs d'opération. Chacun de ces identificateurs est connu du système, il représente une opération possible sur l'objet de type TYPE_OBJET. Cet ensemble est fonction du type d'objet TYPE_OBJET. Si le type TYPE_OBJET est :

- FENETRE, l'ensemble est inclus dans ENS_ID_OP_FENETRE,
- FENETRE_EDITEUR_TEXTE, l'ensemble est inclus dans ENS_ID_OP_-FENETRE_EDITEUR_TEXTE,
- MENU, l'ensemble est égal à ENS_ID_OP_MENU,
- BOITE_A_MESSAGES, l'ensemble est égal à ENS_ID_OP_BOITE_A_-MESSAGES.

- *résultats* :

Un objet de type TYPE_OBJET est créé, son nom est NOM_OBJET. Pour un utilisateur, les opérations possibles sur cet objet appartiennent à l'ensemble des identificateurs ACTIONS_UTILISATEUR.

- *post-conditions* :

Pour chacun des types d'objet composant le type d'objet TYPE_OBJET, le module crée un objet. Ces objets sont associés à l'objet de nom NOM_OBJET. Ces objets sont créés en fonction des identificateurs d'opération appartenant à l'ensemble ACTIONS_UTILISATEUR. Si pour l'objet de type TYPE_OBJET, l'ensemble ACTIONS_UTILISATEUR comprend l'identificateur d'opération suivant :

- DEPLACER (DETRUIRE) : cela signifie que l'objet de type BARRE_TITRE composant l'objet de nom NOM_OBJET, dispose d'une zone "boîte à sélection gauche" ("boîte à sélection droite"),
- MONTER (DESCENDRE) : cela signifie que l'objet de type BARRE_TITRE composant l'objet de nom NOM_OBJET, dispose d'une zone "titre" ("fond"),
- PAGE_PRECEDENTE (SUIVANTE, DEBUT_TEXTE, ou FIN_TEXTE) : cela signifie que l'objet de type RADIO_BOUTONS composant l'objet de nom NOM_OBJET, offre les sélections "page précédente" ("page suivante", "début texte", ou "fin texte"),
- ENVOYER_COMMANDE (ANNULER_COMMANDE, AIDE_COMMANDE, ou DESC_PAR_COMMANDE) : cela signifie que l'objet de type EDITEUR_BOUTONS composant l'objet de nom NOM_OBJET, offre les sélections "envoyer" ("annuler", "aide", ou "description paramètres"),
- EDITION_TEXTE : cela signifie qu'un objet de type EDITEUR_TEXTE compose l'objet de nom NOM_OBJET. Le type TYPE_OBJET ne peut être que FENETRE_EDITEUR_TEXTE.

Si pour un objet de type FENETRE ou FENETRE_EDITEUR_TEXTE, ACTIONS_UTILISATEUR comprend l'identificateur d'opération :

- SELECTION : cela signifie qu'un objet de type MENU ou BOITE_MESSAGES est associé à l'objet de nom NOM_OBJET.

3) Module d'affichage des objets : AFFICHAGE-DES-OBJETS

a) But

Ce module est chargé de l'affichage des objets à l'écran. En effet le module précédent, CREATION-OBJETS crée les objets mais ne les affiche pas à l'écran.
Ce module est également chargé du réaffichage d'un objet lorsque celui-ci est "exposé" (cfr. 2.7) après avoir été partiellement ou entièrement caché par un autre objet.
Ce module doit également tenir compte de la composition d'un objet par d'autres objets.

b) Spécification externe

- *arguments* :

NOM_OBJET : nom de l'objet
POSITION : position écran du coin supérieur gauche de l'objet affiché

- *pré-conditions* :

NOM_OBJET identifie un objet existant.
POSITION est inclus dans les limites de l'écran.

- *résultat* :

L'objet identifié par NOM_OBJET est affiché à l'écran. Son coin supérieur gauche est situé à la position écran POSITION.

- *post-conditions* :

Sont également affichés, les objets composant l'objet de nom NOM_OBJET. L'affichage de cet objet a pour conséquence de cacher tous les objets qui occupent partiellement ou entièrement la même surface d'affichage.
Si besoin est, POSITION est modfié de telle sorte que l'objet affiché de nom NOM_OBJET ne dépasse pas les limites de l'écran.

4) Module de saisie : SAISIE

a) But

Ce module est chargé de la saisie à l'écran des actions effectuées par un utilisateur. Celui-ci dispose du clavier et de la souris pour interagir avec le système. A l'aide de la souris, l'utilisateur peut sélectionner un objet et effectuer une opération sur cet objet comme par exemple, déplacer un objet ou sélectionner une partie de cet objet. A l'aide du clavier, l'utilisateur peut également effectuer une opération (mode expert), ou introduire du texte.
Ce module est chargé d'analyser la saisie c'est-à-dire de faire le lien entre ce qui est saisi et l'action utilisateur correspondante. Par exemple, au relâchement du bouton droit de la souris sur un objet de type menu, correspond l'action utilisateur sélection avec les arguments que sont la sélection du menu, le type menu, et le nom de l'objet.

b) Spécification externe

- *arguments* :

MODE_SAISIE	:	le mode de saise
TYPE_OBJET	:	le type de l'objet

- *pré-conditions* :

MODE_SAISIE appartient à l'ensemble des types suivant :
mode-commande : ce mode correspond à l'entrée de commandes à l'aide du clavier ou de la souris,
mode-textuel : ce mode correspond à l'entrée de texte à l'aide du clavier ou de la souris.

TYPE_OBJET identifie le type d'un objet. Ce type est connu du système, il appartient à ENS_TYPE_OBJET.

- *résultats* :

NOM_OBJET : le nom de l'objet sélectionné par l'utilisateur
ACTION_SAISIE : l'identificateur de l'opération (action utilisateur) souhaitée par l'utilisateur
ARGUMENTS_ACTION_SAISIE : la liste des arguments de l'opération saisie

- *post-conditions* :

NOM_OBJET est le nom d'un objet de type TYPE_OBJET sur laquelle une action utilisateur ACTION_SAISIE est effectuée, avec les arguments ARGUMENTS_ACTION_SAISIE.
ACTION_SAISIE est l'identificateur d'opération. Cet identificateur est connu du système, il appartient à ENS_ID_OPERATION.
ARGUMENTS_ACTION_SAISIE est la liste des arguments supplémentaires nécessaires pour effectuer l'opération identifiée par ACTION_SAISIE. Ces arguments sont de types différents suivant l'identificateur d'opération. Pour les identificateurs d'opérations suivants :

- MONTER, DESCENDRE, DETRUIRE : les arguments sont la pile des objets graphiques, et un écran,
- DEPLACER : les arguments sont une position écran, la pile des objets graphiques et un écran,
- SELECTION : les arguments sont une position, et la file de l'application contenant les commandes effectuées par l'utilisateur qui doivent être traitées par l'application,
- ENVOYER_COMMANDE (ANNULER, AIDE, DESC_PAR) : l'argument est le texte saisi.

4.3.5 Quatrième étape : le codage

Nous ne présentons que la phase de codage global de cette quatrième étape. Le codage détaillé se trouve en annexe1 et en annexe2. A nouveau, ne disposant pas d'une architecture physique nous avons dû la reconstruire à partir du système existant.

4.3.5.1 Architecture physique du système

Nous n'avons pas à proprement parlé fait une architecture physique du système. Nous avons simplement en fonction du temps disponible essayé de voir les appels de fonctions entre les différents fichiers LE_LISP.
Le résultat est un graphe dans lequel chaque noeud est représenté par une boîte dans laquelle sont indiqués le nom du fichier ainsi que les "grandes" fonctions qu'il contient. Les flèches représentent une relation de type "appel" entre les fonctions de deux fichiers liés. Néanmoins, une correspondance peut être observée avec l'architecture logique quant à la disposition en niveaux des différentes boîtes. Cette architecture "physique" est présentée à la figure 4.3.16.

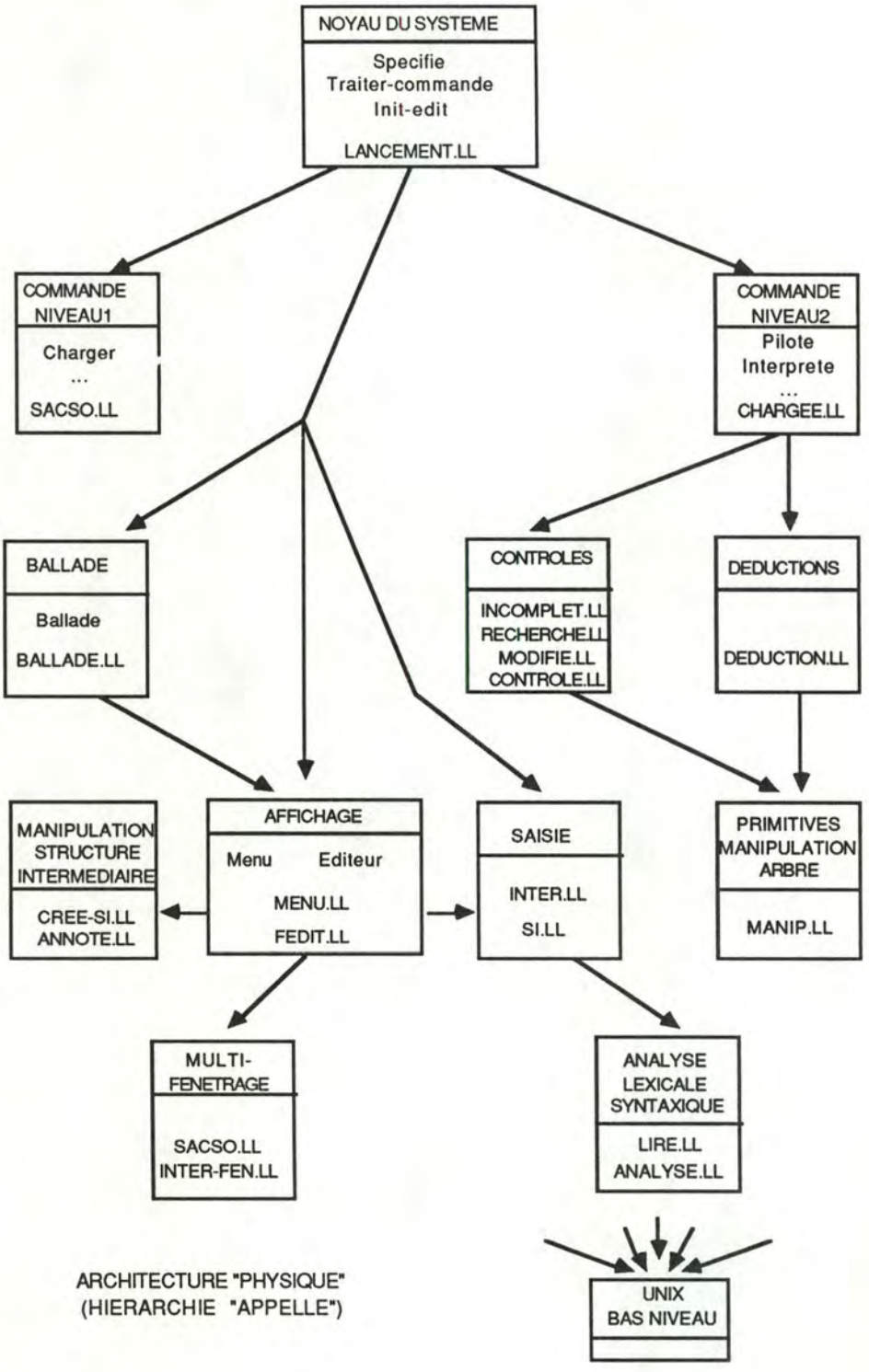


Figure 4.3.16 : l'architecture "physique" de SACSO

4.3.5.2 Découpe physique de la couche : le niveau 3

1) Introduction

Au niveau physique les opérations sur les objets identifiées précédemment (cfr. les figures 4.3.12, 4.3.13, 4.14), sont implémentées par un module physique (primitive). Nous avons

implémenté ces primitives avec X Windows en langage C. Malheureusement, X Windows n'offre que des procédures de bas niveau. On peut citer, par exemple, le fait qu'il ne gère pas le contenu des fenêtres, et qu'il n'est pas possible de manipuler directement un menu. En effet, la création d'un menu nécessite d'abord le remplissage d'un record en précisant le nombre d'items que doit comporter le menu, la définition du fond (background) qui doit être utilisé, la définition de la police de caractères à utiliser, etc.. Nous avons, pour cette raison écrit des primitives de plus haut niveau qui implémentent les opérations nécessaires sur chaque objet.
Les spécifications complètes et détaillées sous forme pré-post de ces procédures se trouvent en annexe 1.
Nous allons, pour l'instant, nous attarder sur quelques concepts importants de cette couche.

2) Concepts importants de la couche

A chaque opération identifiée pour les objets, correspond généralement une primitive la réalisant. Cependant, certaines opérations offertes directement par X Windows, ne nécessitent pas une primitive particulière dans notre couche. C'est le cas, par exemple, de l'obtention d'une sélection dans un éditeur ou dans un menu. Il en est de même pour le menu qui s'efface automatiquement après obtention de la sélection.
Une attention particulière doit être portée à **la gestion des événements**. L'utilisateur interagit avec les différents objets par l'intermédiaire de la souris et du clavier. Ces interactions provoquent la génération d'événements par le gestionnaire de clavier et de souris de X Windows.
Ces événements peuvent être partagés en différentes classes d'événements correspondant chacune à un objet particulier. Une gestion des événements par type d'objet est donc requise : traiter les événements du type menu ("Traiter- menu"), du type boîte à messages ("Traiter-boîte-messages"), etc..
Ces différentes gestions particulières composent la gestion globale des événements vue comme une procédure composée de sous-procédures chargées chacune de traiter une classe particulière d'événements. Chaque sous-procédure peut encore être décomposée afin de traiter les différents événements spécifiques à une même classe d'événements. Par exemple, pour l'objet de type menu, la procédure "Traiter-menu" doit traiter le premier item, traiter le second item, etc..
Cette gestion des événements permet de particulariser l'interaction de l'utilisateur avec un objet. Suivant l'emplacement ("souris") de l'utilisateur dans tel ou tel objet, le traitement des événements sera différent. La gestion des événements est fonction du contexte. Par exemple, pour l'objet menu, nous prenons en compte des événements "souris" mais également des événements "clavier" afin de permettre des racourcis. La possibilté de rendre "actives" ou "inactives" certaines "zones" d'un objet accentue encore cette caractéristique "fonction du contexte" de la gestion des événements.

Au niveau de la boîte à outils étendue, nous ne pouvons proposer qu'un **squelette** de gestion des événements. Ce squelette doit être particularisé pour une application. C'est au niveau de celle-ci que nous associons une sémantique à l'événement. Par exemple, dans le cas d'un menu, à la sélection d'un certain item est associée la commande "DUPLIQUER".

4.3.5.3 Remplacement du gestionnaire de multi-fenêtrage de SACSO (du niveau 4)

La couche construite en C est utilisée pour construire un gestionnaire de multi-fenêtrage. Pour pouvoir l'intégrer dans SACSO, ce gestionnaire doit être écrit en LE_LISP. Nous devons donc, pour sa construction, pouvoir utiliser en LE_LISP, les primitives C de la couche. La description complète et détaillée de la manière dont nous avons procédé se trouve en annexe 2. En voici le principe : les fonctions C ont été déclarées en LE_LISP au moyen de la fonction LE_LISP (Defextern <symb> <ltype> <type>) qui permet d'associer dynamiquement un module externe avec une fonction LE_LISP. <symb> est le nom d'un module externe qui va devenir une nouvelle fonction LE_LISP, <ltype> est la liste des types des arguments et <type> le type de la valeur retournée. Ces types sont l'un des symboles suivants :

- T pour des pointeurs LE_LISP,
- FIX pour des nombres entiers,
- FLOAT pour des nombres réels,
- STRING pour des chaînes de caractères,
- VECTOR pour des adresses de vecteurs.

Pour appeler les primitives C en LE_LISP il suffit d'utiliser le nom des primitives définies en C et d'ajouter devant ce nom le caractère "_" (underscore). Par exemple la primitive "TRAITER_EVENEMENTS" définie en C sera appelée en LE_LISP par l'appel (_TRAITER_EVENEMENTS).

Chacun des objets identifiés en sections 4.3.3.2 et 4.3.3.3 (fenêtre, fenêtre éditeur de texte, menu, ...) et repris en section 4.3.4, fait l'objet d'un "record" CEYX et d'une série d'opérations associées (monter, descendre, ...). A chaque objet, nous pouvons envoyer des messages correspondant aux opérations à effectuer. Un tel record comprend une série de champs destinés à identifier l'objet qu'il représente, et une série de champs destinés à identifier les objets qui le composent.

4.3.5.4 Intégration du gestionnaire de multi-fenêtrage : XSACSO

Le système SACSO est écrit entièrement en LE_LISP. Nous com- mencons par expliquer brièvement le fonctionnement du noyau de SACSO. Ensuite, nous verrons en quoi la présence de deux "environnements" différents C et LE_LISP peut poser problème.

1) Fonctionnement du noyau de SACSO

Le système SACSO est entièrement écrit en LE_LISP. Le programme principal se présente grossièrement comme une boucle principale dans laquelle à chaque itération une commande est traitée. La commande "quitter" du système permet de sortir de cette boucle. Dans la suite, nous parlerons de la "boucle LE_LISP".
Nous avons également vu que dans notre gestionnaire, la gestion des événements consiste en une boucle dans laquelle des classes d'événements sont traitées. Dans la suite, nous parlerons de la "boucle C".
Un des principaux problèmes posé par l'intégration du nouveau gestionnaire est celui de l'intégration de la gestion par événements.

2) Deux environnements, C et LE_LISP

Nous avons vu que l'intégration de fonctions C en LE_LISP consiste en la déclaration en LE_LISP de modules externes. Par invocation d'une fonction LE_LISP définie en "defextern", le sytème nous permet de lancer l'exécution d'une fonction C. Pendant cette exécution, le système se situe dans un "environnement C". Il ne retournera à l'environnement de l'appelant c'est-à-dire "l'environnement LE_LISP" qu'en fin de la fonction appelée, auquel cas il quitte également "l'environnement C". Depuis LE_LISP, le système peut appeler C par invocation d'une fonction définie en "defextern". Depuis C, il est possible d'appeler LE_LISP par invocation d'une fonction C "lispcall". Dès lors, l'intégration de la gestion par événements dans le système SACSO peut se concevoir de deux manières différentes :

- l'une se conçoit à partir d'un programme principal écrit en langage C. Celui-ci gère les différents événements, et fait appel aux primitives LE_LISP de SACSO implémentant les différentes commandes de l'utilisateur,
- l'autre consiste en un programme principal écrit en langage LE_LISP qui fait appel aux primitives C pour la gestion des événements.

Première solution : le programme principal dans l'environnement C

Lorsque par exemple l'utilisateur choisit dans le menu la commande DUPLIQUER, ceci provoque un certain événement auquel un certain traitement est associé. Dans notre cas le traitement consiste à exécuter la commande LE_LISP DUPLIQUER (appel de LE_LISP depuis C) c'est-à-dire qu'on sort de l'environnement C pour entrer dans l'environnement LE_LISP. Ceci est illustré dans la figure 4.3.17 dans laquelle les flèches représentant une séquence temporelle.

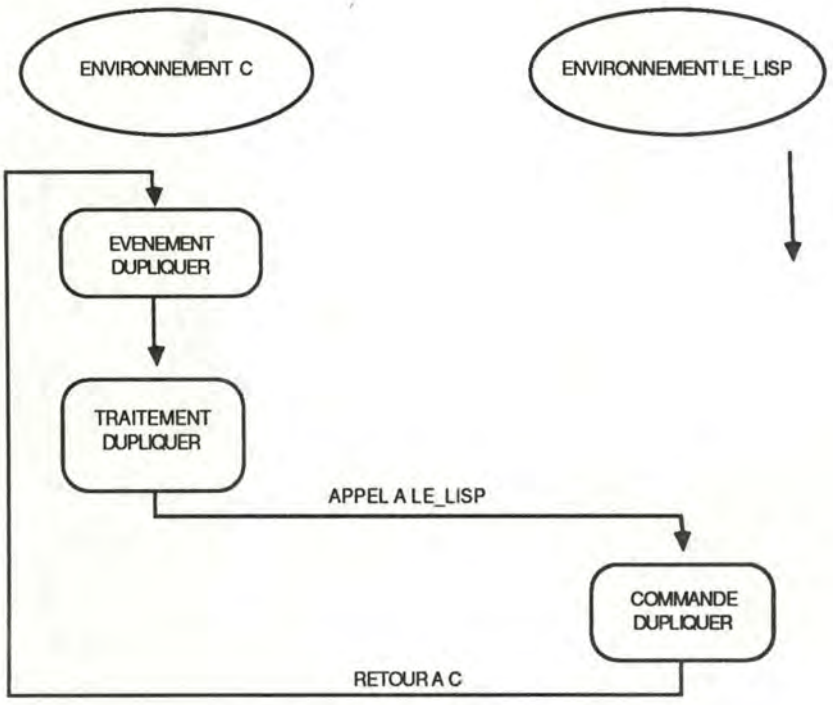


Figure 4.3.17 : première solution - commande non suspendue

Pour permettre à l'utilisateur de suspendre l'exécution d'une commande pour en exécuter une autre, nous devons avoir la possibilité de retourner dans l'environnement C et de savoir quand dans l'environnement LE_LISP, il est nécessaire de retourner dans C.
Un appel à la boucle C permet de passer dans l'environnement C. Reste le problème de savoir où placer cet appel. Supposons le problème résolu et observons le comportement de cette solution lorsque l'utilisateur suspend par exemple l'exécution de la commande DUPLIQUER pour lancer la commande DETRUIRE. Notons en toute logique, qu'en fin de l'exécution de la commande DETRUIRE le système devrait retourner à l'endroit de la suspension de la commande DUPLIQUER. Cette solution est illustrée à la figure 4.3.18, les flèches représentant une séquence temporelle.

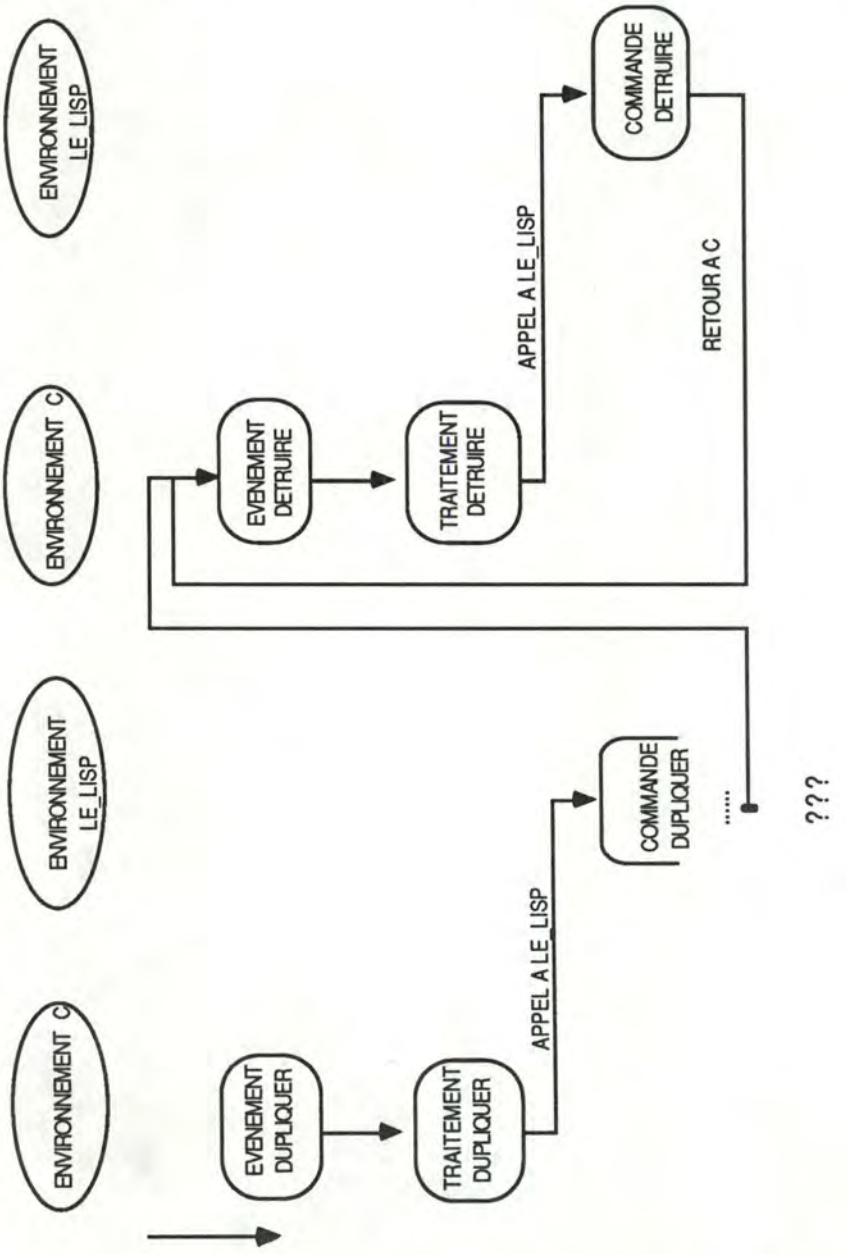


Figure 4.3.18 : première solution - commande suspendue

Ce schéma montre clairement qu'en fin de la commande DETRUIRE le système retourne au deuxième appel de la boucle C et non pas dans l'environnement LE_LISP. La commande

DUPLIQUER ne se terminera donc jamais. Il y a donc "empilement" des environnements et non retour à la commande suspendue.

Deuxième solution : le programme principal dans l'environnement LE_LISP

La deuxième solution est celle que nous avons adoptée. Elle utilise également la boucle C mais à la différence que la boucle LE_LISP représente le programme principal. Dans cette optique la boucle C appelé depuis LE_LISP, doit être vue comme une fonction qui après évaluation indique au programme appelant ce que l'utilisateur veut exécuter. Pour cela la boucle C renvoit une valeur identifiant la commande de l'utilisateur.
Reprenons l'exemple de la commande DUPLIQUER exécutée sans suspension par une autre commande et faisons l'hypothèse que l'on se trouve dans l'environnement C. L'utilisateur choisit dans le menu la commande DUPLIQUER. Ensuite l'environnement C est quitté en renvoyant à la fonction appelante, en l'occurence la boucle LE_LISP, la chaîne de caractères "DUPLIQUER". La commande DUPLIQUER est exécutée en LE_LISP. Le schéma de cette solution est présenté à la figure 4.3.19.
Le déroulement est simple, il suffit d'évaluer la boucle C pour connaître la commande de l'utilisateur et puis de lancer l'exécution de cette commande.
Considérons maintenant l'exemple de la commande DUPLIQUER suspendue pour lancer la commande DETRUIRE. Nous voyons que cette solution a pour avantage de permettre "l'empilement" des commandes en évitant "l'empilement" des environnements et le problème du non-retour à la commande suspendue. Cette solution est illustrée à la figure 4.3.20.
Ce schéma montre clairement que l'exécution de la commande DUPLIQUER peut être suspendue pour lancer une autre commande et être sûr qu'en fin d'exécution de cette dernière, l'utilisateur se retrouve au point de suspension.

Nous avons supposé au début de la présentation de cette deuxième solution que nous nous situions dans l'environnement C. Comme cette solution suppose que le programme principal se trouve dans l'environ- nement LE_LISP, nous devons maintenant lever notre hypothèse et voir quand placer l'appel à la boucle C dans la boucle LE_LISP. L'idée a été de remplacer ce qu'on pourrait appeller "les fonctions d'interaction", c'est-à-dire les fonctions qui permettent de savoir ce que veut l'utilisateur, par un appel à cette boucle C (voir les fonctions LE_LISP TRAITER-COMMANDE, COMMANDE-SAISIE, EDIT en annexe pour plus de détails).

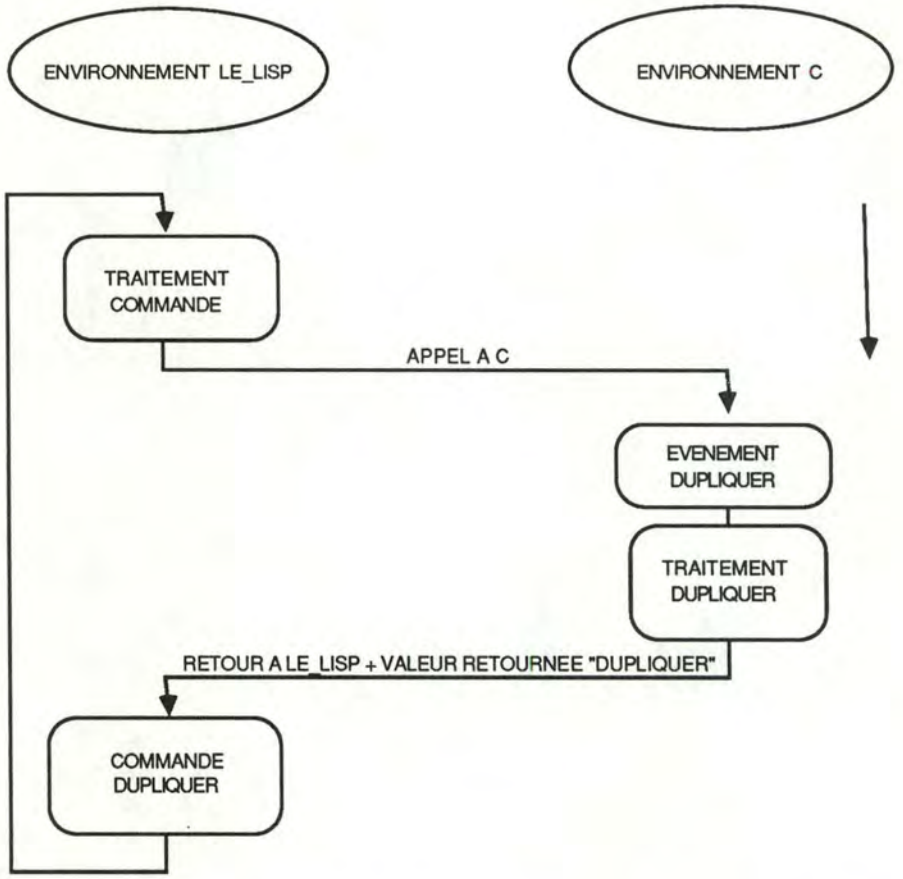


Figure 4.3.19 : deuxième solution - commande non suspendue

4.4 Conclusion

La nouvelle interface que nous avons construite pour SACSO présente encore les défauts suivants :

- dans certains cas, il faudrait offrir un feedback à l'utilisateur. Ce feedback serait particulièrement nécessaire lorsque LE_LISP fait son "garbage collecting". Dans ce cas, le temps de réponse du système devient mauvais et il faudrait donc signaler à l'utilisateur par un message que le système est "indisponible" pour quelques instants,

- certaines commandes, qui sont encore introduites par l'utilisateur au moyen du clavier pourraient l'être au moyen d'un menu.

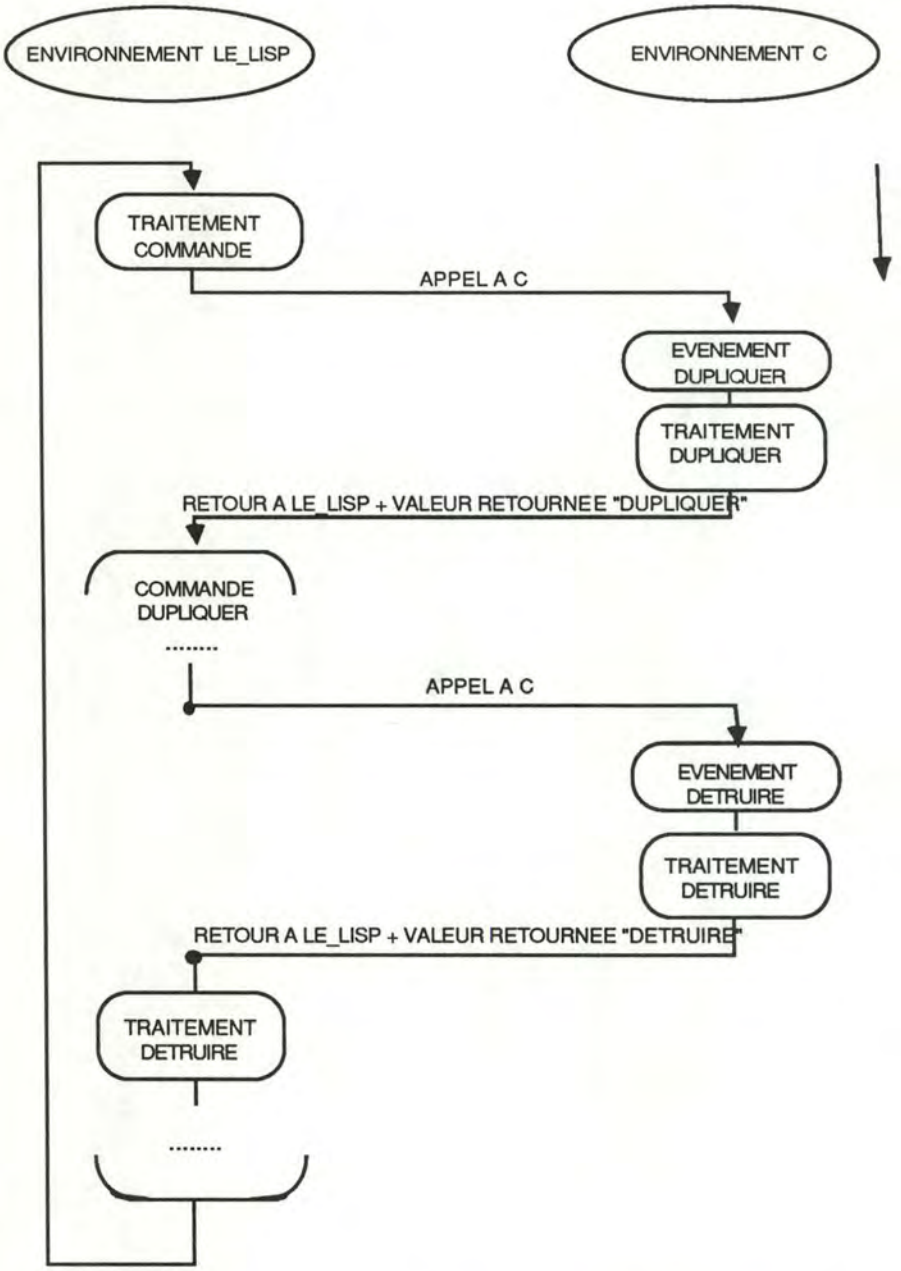


Figure 4.3.20 : deuxième solution - commande suspendue

Cependant, le système XSACSO présente par rapport au système SACSO, les avantages suivants :

1) au niveau de l'interface :

- l'utilisation de **la souris** permet de se "ballader" dans tout l'écran de manière plus naturelle que par l'emploi des touches de contrôle du clavier. La souris permet également de se positionner n'importe où dans un texte d'édition, remplaçant agréablement les commandes d'édition de texte du système SACSO. En dernier lieu, nous pouvons dire que l'utilisation de la souris permet de diminuer le nombre de commandes que l'utilisateur doit connaître,

- l'utilisation de **commandes "raccourcis"** permet d'adapter le système à une catégorie d'utilisateurs experts. L'utilisation de raccourcis permet entre autres de court-circuiter le schéma classique de traitement d'une commande (édition de la commande, vérification de la commande, lancement de la commande) en évitant la phase d'édition de la commande,

- l'utilisation des **possibilités graphiques** plus étendues d'une station de travail de type SUN permet de construire une interface plus agréable (graphisme plus élaboré, taille de l'écran plus grande, etc.).

2) au niveau des performances :

Considérons le temps de réponse d'un système défini comme suit :

"le temps séparant le moment où l'utilisateur lance l'exécution de sa commande de celui où l'utilisateur voit le premier élément de réponse".

En considérant une telle définition, nous ne pouvons pas affirmer que le système XSACSO offre un meilleur temps de réponse que le système SACSO, puisque nous n'avons pas réalisé de tests de performance. Cependant, nous pouvons souligner les faits suivants qui dans une certaine mesure, font partie des critères d'efficacité d'un système interactif :

- XSACSO donne la possibilité d'introduire plus rapidement et avec moins d'erreurs, les différentes commandes du système. En effet, l'utilisateur dispose de zones à sélections (menu, éditeur à boutons) qui lui permettent par simple clic d'un bouton de la souris, d'introduire une sélection. La sélection est plus rapide puisque l'utilisateur n'est plus obligé comme dans le système SACSO, de se positionner dans une fenêtre spécifique au menus. Ceux-ci étant directement accessibles n'importe où dans l'écran par simple clic d'un des boutons de la souris. Par l'utilisation de raccourcis et de menus, l'utilisateur évite la phase d'édition de la commande et réduit considérablement la phase de vérification de la commande.
  Nous pouvons donc dire que XSACSO permet de diminuer le nombre d'erreurs dues à l'emploi du clavier (fautes de frappe, texte mal introduit, etc.). Par conséquent, la gestion des erreurs s'en trouve simplifiée, et les performances améliorées,

- XSACSO offre des primitives d'affichage qui sont plus rapides. Ceci permet de diminuer le temps d'affichage défini comme suit :
  "le temps séparant l'affichage du premier élément de réponse de la commande, de celui du dernier élément de réponse".

Les remarques concernant les étapes que nous avons suivies pour la construction d'une nouvelle interface pour SACSO sont les suivantes :

- nous situant dans une étape de maintenance, certains documents comme par exemple l'architecture logique et l'architecture physique nous étaient indispensables pour voir la structure globale du système et analyser les répercussions des modifications que nous envisagions de faire.

Ces documents n'étant pas disponibles nous avons dû les reconstruire à partir du système existant,

- dans la démarche que nous avons suivie, telle qu'exposée dans cette section, n'apparaît pas le fait que nous avons procédé par prototypage. Il nous semble en effet que le prototypage constitue une bonne approche pour la construction d'une interface, car il est difficile de savoir exactement ce que désire l'utilisateur. Mieux vaut d'abord spécifier les fonctionnalités de l'interface pour l'utilisateur, puis montrer la dynamique de cette interface sans insister sur les différentes touches qui permettront de réaliser telle ou telle action mais en montrant les différentes possibilités qu'a l'utilisateur pour lancer une action. Ensuite, il faut construire un prototype et le présenter aux utilisateurs pour tenir compte de leurs critiques et remarques et pour observer les difficultés qu'ils éprouvent à utiliser le système avec l'interface. Cela entraîne de nombreux retours en arrière au niveau de la spécification des fonctionnalités et de la dynamique jusqu'à ce qu'un prototype satisfaisant l'utilisateur soit réalisé,

- les outils que nous avons présentés dans la première partie ont exercé une certaine influence sur notre démarche. Il était en effet très difficile, connaissant X Windows, de ne pas déjà parler de concepts de X Windows dans l'étape de spécification du gestionnaire de multi-fenêtrage. Toutefois au niveau de la spécification du gestionnaire de multi-fenêtrage nous utilisons des concepts logiques de menus, fenêtres et c'est pour cette raison que nous n'avons pas identifié plus précisément tous les objets interactifs et les opérations possibles sur ces objets interactifs au niveau de l'étape de spécification du gestionnaire de multi-fenêtrage.

Ce chapitre termine la partie 2 consacrée à la modification du multi-fenêtrage de SACSO. Nous allons entamer la partie 3 qui présente un outil de visualisation générique d'objets formels. La conception de cet outil résulte des besoins graphiques de SACSO exposés dans le chapitre 5.

Outils graphiques pour environnements logiciels dirigés par la syntaxe

(volume 2)

Christophe Paquet et Marc Paring

Promoteur
Monsieur **Axel van LAMSWEERDE**

Mémoire présenté pour l'obtention du grade
de Licencié et Maître en Informatique
par

Christophe PAQUET et **Marc PARING**

Année académique 1987 - 1988

PARTIE III

UN OUTIL GENERIQUE DE VISUALISATION GRAPHIQUE D'OBJETS FORMELS

Chapitre 5

Présentation du problème

5.1 Introduction

L'utilisation de graphisme pour représenter des objets manipulés par un utilisateur à l'écran constitue un moyen d'améliorer l'interface homme-machine. Cette tendance se reflète de plus en plus dans les nouvelles interfaces d'ordinateurs. Le Macintosh d'Apple en est probablement le meilleur exemple.

Dans ce même ordre d'idées, une représentation graphique de l'arbre des types (opérations) d'un type (opération) d'une spécification SACSO est envisagée. Nous présentons en section 5.2 les objectifs de ce projet et en section 5.3 les problèmes à résoudre dans ce cadre.

5.2 Les objectifs

Dans le système SACSO, il est difficile d'avoir une vue globale de la spécification en cours de construction surtout lorsque le pilote est utilisé. Il est difficile de se souvenir exactement de ce qui a été fait et de ce qui reste à faire. Partant de ce problème, l'idée nait d'avoir un outil qui permet d'afficher à l'écran une représentation graphique de l'arbre des types (opérations) d'un type (opération). L'utilisateur grâce à cet outil pourrait visualiser graphiquement la structure d'un type. Cette visualisation peut se faire en cours de construction de la spécification. A titre d'exemple, on se référera à la forêt des types et la forêt des opérations de la spécification MULTIFENETRAGE (cfr 4.3.3.2 figures 4.3.2 et 4.3.3)

Supposons que la représentation graphique des constructeurs de types de SACSO soit la suivante :

- pour la table (cfr figure 5.2.1),

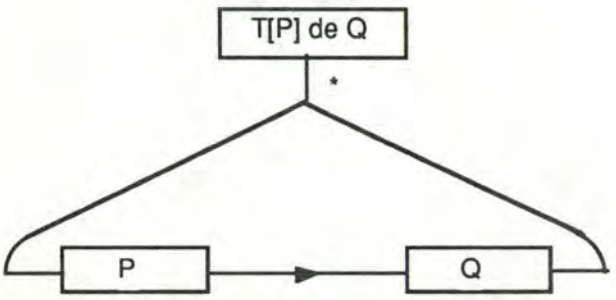


Figure 5.2.1 : représentation graphique de la table

- pour la suite (cfr figure 5.2.2),

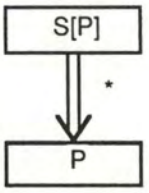


Figure 5.2.2 : représentation graphique de la suite

- pour l'ensemble (cfr figure 5.2.3),

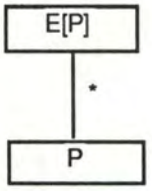


Figure 5.2.3 : représentation graphique de l'ensemble

- pour le produit cartésien (cfr figure 5.2.4),

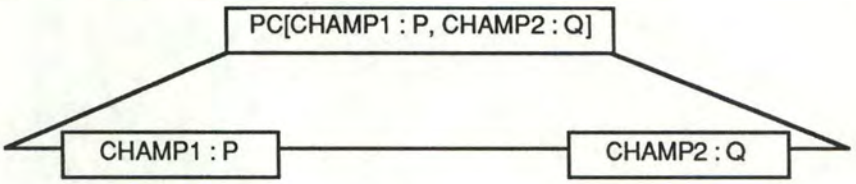


Figure 5.2.4 : représentation graphique du produit cartésien

- pour l'union disjointe (cfr figure 5.2.5),

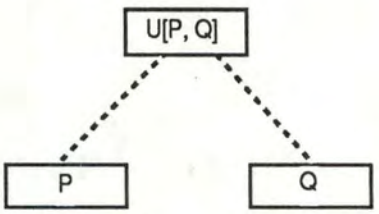


Figure 5.2.6 : représentation graphique de l'union disjointe

Dès lors, la représentation graphique de l'arbre des types du type CONTINUUM de la spécification présentée à la section 1.8 serait la suivante (cfr figure 5.2.7)

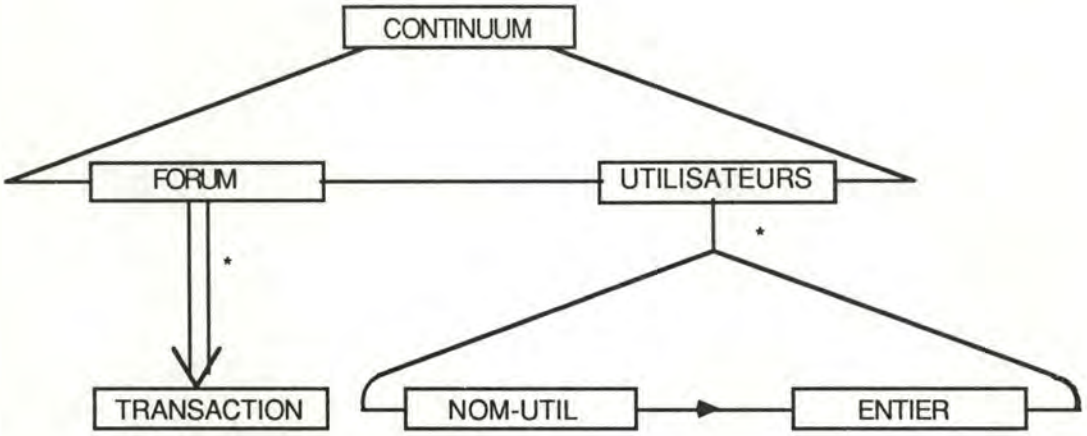


Figure 5.2.7 : représentation graphique de l'arbre des types du type continuum de la spécification CONTINUUM

Sur cette représentation graphique, l'utilisateur doit pouvoir réaliser un certain nombre d'opérations. Il doit pouvoir aller cliquer dans les boîtes RECTANGLE (qui représentent des types) et obtenir une information plus détaillée (sous forme de texte) dans une autre fenêtre. En gros, les informations textuelles obtenues dans le système doivent pouvoir être obtenues à partir de la représentation graphique de ce type. L'objectif est d'ajouter à la représentation textuelle existante, une autre représentation graphique et de pouvoir passer de l'une à l'autre.

5.3 Problèmes à résoudre

Pour savoir ce que l'on veut visualiser, il faut d'abord expliciter le concept de type. Rappelons qu'un type SACSO (cfr section 1.3) est soit :

- un type de base,
- un type de l'environnement (bibliothèque),
- un type de la spécification.

Les problèmes qui se posent pour la représentation graphique de l'arbre des types pour un type donné de la spécification sont les suivants.

5.3.1 Représentation d'un type

Un type est normalement défini par son nom, l'objectif associé, la structure du type, l'invariant et les opérations associées à ce type. Dans la visualisation graphique, on pourrait représenter uniquement le nom du type et sa structure schématisée, ou bien lui adjoindre en plus les opérations associées. Toutefois si l'on adjoint au type la liste des opérations associées, la structure globale du type n'est plus aussi claire ; d'autre part, la taille de l'écran étant limitée, il risque d'être impossible de tout représenter.

5.3.2 Profondeur d'un arbre de types

Il se pose un problème de représentation de la structure d'un type dans la mesure où un type de la spécification peut par exemple avoir une arborescence très profonde. Une solution consiste à ne pas descendre trop bas dans la représentation graphique et se limiter à la représentation de l'arborescence jusqu'aux types de l'environnement. En effet, les types de

l'environnement peuvent être très complexes et donc avoir une arborescence très profonde. Dans le système existant, il n'est pas possible d'avoir plus de détails sur un type de l'environnement que jusqu'à un certain niveau de détail car pour des raisons de place mémoire l'environnement n'est pas entièrement chargé en mémoire centrale. D'autre part cela n'aurait pas beaucoup de sens de redétailler chaque fois la structure d'entités prédéfinies.
On pourrait donc déduire la règle suivante :

règle 1 : Descendre jusqu'aux types de l'environnement sauf si la profondeur est trop importante pour la taille de la fenêtre auquel cas des règles plus fines doivent être appliquées (cfr règle 2 ci-après)

5.3.3 Largeur et hauteur d'un arbre de types

L'arbre des types peut avoir une largeur et une hauteur importantes. Il faut donc ne représenter qu'une partie de l'arbre tant en largeur qu'en hauteur. Une solution consiste dans le cas où il n'est pas possible d'afficher l'arbre dans toute sa largeur et dans toute sa hauteur, de laisser à l'utilisateur le soin de demander plus de détail ce qui lui permettrait de visualiser le reste de l'arbre dans une autre fenêtre ou bien de diminuer la taille de l'arbre ce qui permettrait de visualiser la structure globale de l'arbre.
Ceci donne la règle suivante :

règle 2 : afficher l'arbre du type dans toute sa largeur et toute sa hauteur. Dans le cas où ceci n'est pas possible, représenter ce qu'il est possible de représenter et mettre trois points ou bien un nom d'holophraste à la place des parties de l'arbre non représentées ; laisser à l'utilisateur la possiblité de demander plus de détails concernant les parties non représentées.

5.3.4 Association de la représentation graphique à un arbre abstrait

Il faut déterminer l'information qui apparaîtra à l'écran lorsque l'utilisateur effectuera une action sur la représentation graphique. Dans un premier temps, l'information fournie dans le système existant sera présentée. Dans un deuxième temps, on pourrait imaginer de laisser à l'utilisateur un certain contrôle c'est-à-dire une sélection de ce qu'il veut comme information. Par exemple, au lieu d'avoir pour un type son objectif, son invariant, ses opérations associées etc..., l'utilisateur pourrait demander de n'avoir que les opérations ou bien que l'objectif.

D'autre part, la représentation graphique de l'arbre d'un type peut être modifée en cours de construction ce qui impose de rattacher la représentation graphique à l'arbre abstrait. Il faut de toute façon que les deux représentations soient constamment en cohérence.

5.3.5 Aperçu de la solution proposée

Ces problèmes auraient pu être résolus de manière spécifique à SACSO. Nous avons adopté une approche plus générale. Plutôt que de nous limiter à une solution ne permettant que de représenter l'arbre des types d'un type d'une spécification, la solution présentée est plus générale. Elle permettra de préciser quels sont les objets d'un arbre abstrait qu'il faut représenter graphiquement, la relation qui lie ces objets ainsi que la représentation graphique souhaitée des objets et des relations (voir 6.5 et 6.6).

La solution que nous proposons est générique selon plusieurs aspects :

- l'outil de visualisation est générique par rapport à un **langage d'interface graphique** défini grâce à un méta-langage. L'outil de visualisation peut donc être instancié à un langage d'interface graphique particulier. Cette généricité est assurée au moyen de la table des objets et des relations qui contient la "définition" du langage d'interface graphique. Un langage d'interface graphique (L.I.G.) est constitué d'un ensemble de définitions de représentations de concepts sémantiques. Un L.I.G. permet la construction correcte de toute représentation graphique d'une combinaison d'occurrences de concepts sémantiques repris dans l'ensemble de ces concepts sémantiques. Un langage d'interface graphique est décrit par une grammaire se présentant sous la forme d'un ensemble de définitions de représentations pour un concept sémantique.
  Une définition de représentation est donnée par trois éléments :

  - le concept sémantique (exemple : un type SACSO),
  - la représentation graphique souhaitée pour représenter toute occurrence de ce concept sémantique (exemple : rectangle),
  - le mode de composition spatiale souhaité (exemple : horizontal, vertical ...)
    - soit entre plusieurs occurrences d'un même concept sématique,
    - soit entre une occurrence d'un concept sémantique et une occurrence d'un autre concept sémantique.

  Nous proposons un méta-langage (cfr chapitre 6) permettant de définir un L.I.G. Un L.I.G. peut alors être utilisé pour décrire la représentation graphique d'objets et de relations (concepts sémantiques) modélisés dans un arbre abstrait (représentant par exemple une spécification SACSO). Cette solution constitue donc une généralisation du cas spécifique de représentation de l'arbre des types d'un type apparaissant dans une spécification SACSO,

- l'outil de visualisation est générique par rapport aux **concepts graphiques** qui sont utilisés pour décrire la représentation des objets et des relations inter-objets. Il est possible d'ajouter de nouveaux concepts graphiques qui peuvent être utilisés dans le méta-langage pour définir un langage d'interface graphique. Cette généricité est assurée par la table des concepts graphiques qui contient la "définition" des différents concepts graphiques,

- l'outil de visualisation est générique par rapport à une **structure physique** d'arbre abstrait. Cette généricité est assurée au moyen de la table des correspondances entre structures logiques et physiques qui contient la "définition" de la structure physique de l'arbre abstrait.

L'utilité et le contenu de ces tables sont expliqués en détail dans le chapitre suivant.

Chapitre 6

Proposition de solution

6.1 Introduction

Dans ce chapitre, nous détaillons la solution générale proposée aux problèmes exposés au chapitre 5. Avant de présenter l'architecture générale de l'outil de visualisation (section 6.3), nous présenterons à la section 6.2 le concept de boîte qui sera utilisé dans la suite de l'exposé. Dans les sections suivantes, les sous-problèmes découlants de l'architecture présentée à la section 6.3 seront exposés ; la section 6.4 donne une brève description de la structure générale de tout arbre abstrait représentant une spécification SACSO qui nous servira d'exemple. Ensuite en section 6.5 et 6.6, nous décrirons respectivement le langage de description d'objets et de relations inter-objets (L.D.O.R.) permettant de décrire les objets et relations à représenter graphiquement et le langage de description graphique. Les différents modules, structures et tables de l'architecture présentée en section 6.3 seront détaillés dans les sections suivantes. Nous commencerons par le module de saisie (section 6.7), pour poursuivre avec le module de décompilation graphique (section 6.8) et les tables (section 6.9). La structure physique de la structure intermédiaire sera présentée en section 6.10, le module d'annotation en section 6.11. Nous terminerons en section 6.12 par le module d'affichage, quelques exemples de structures logiques pouvant être utilisées dans le L.D.O.R. en section 6.13, et finalement quelques algorithmes abstraits en section 6.14.

6.2 Le concept de boîte

En général, les applications manipulent des objets autres que les abstractions définies pour les terminaux (caractères, pixels etc...). Le concepteur d'une interface souhaiterait pouvoir exprimer des requêtes du genre "Afficher ces deux objets l'un en dessous de l'autre". C'est pour cette raison, qu'une représentation intermédiaire est nécessaire.

Le concept de **boîte** [Coutaz 85] permet aux applications d'exprimer des entrées/sorties de haut niveau, orientées-**objet**. Une boîte est une représentation intermédiaire entre les entités internes que l'application doit présenter à l'utilisateur et les informations visibles à l'écran. Une boîte doit pouvoir exprimer des relations structurelles entre objets. Cette expression repose sur une arborescence dans laquelle :

- une feuille contient une information encadrée d'un rectangle imaginaire (boîte terminale),
- un noeud est une boîte, résultat de la composition spatiale de boîtes filles,
- chaque boîte est décorée d'attributs héritables des boîtes parentes, permettant l'expression de relations spatiales et l'affectation de propriétés syntaxiques.

Certains objets ont des liens sémantiques que l'application peut vouloir exprimer visuellement sous forme de relations spatiales. Par exemple, des relations de composition peuvent être représentés par des relations spatiales d'alignement. Pour ce faire, l'application spécifie les relations entre les entités internes à présenter sur l'écran. Cette spécification produit une structure de boîtes qui est ensuite décompilée à l'écran. Cette structure est un arbre décoré.

L'arbre traduit la composition d'entités. Les relations spatiales sont exprimées au moyen d'attributs de composition qui peuvent prendre les valeurs suivantes :

- H : concaténation horizontale des rectangles des boîtes filles,
- V : concaténation verticale des rectangles des boîtes filles,
- HOV (HouV) : concaténation horizontale des boîtes filles si la place disponible est suffisante, concaténation verticale sinon,
- HV (HetV) : concaténation horizontale, puis passage à la ligne lorsque la place disponible ne permet plus la concaténation horizontale d'une boîte fille supplémentaire.

6.3 Architecture générale de l'outil de visualisation

Nous expliquons dans la suite, l'architecture de l'outil de visualisation textuelle (décompilateur) du système SACSO actuel, sur laquelle nous avons calqué notre architecture (voir partie supérieure de la figure 6.3.1).

L'outil de visualisation du système existant est organisé de la manière suivante :

- l'arbre abstrait représente une spécification SACSO,
- le module de décompilation textuelle reçoit en entrée le noeud à décompiler et crée la structure intermédiaire qui a une structure d'arbre basé sur la notion de boîte,
- la structure intermédaire contient tout le texte à afficher sans notion de positions,
- le module d'annotation de la structure intermédiaire textuelle, en fonction de la taille de la fenêtre, crée la structure intermédiaire avec notion de positions ; ce module affecte à chaque noeud à afficher une position (x, y) à laquelle l'objet représenté par le noeud sera affiché à l'écran,
- la structure intermédiaire textuelle avec notion de positions est identique à la structure intermédiaire textuelle sans notion de positions, à ceci près que tous les noeuds ont une position à laquelle les objets qu'ils représentent seront affichés,
- le module d'affichage se contente d'afficher la structure intermédiaire textuelle avec notion de positions dans une fenêtre.

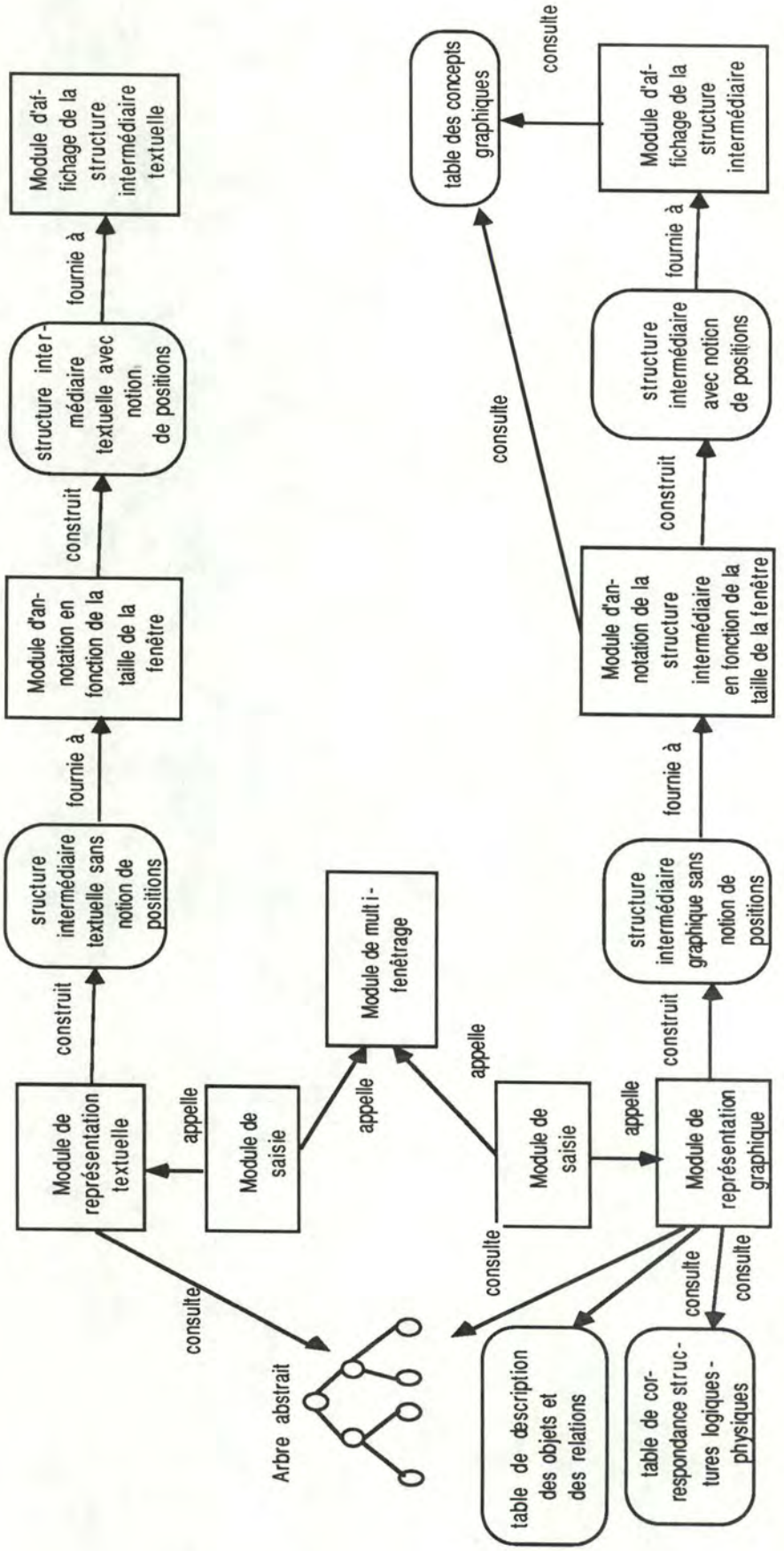


Figure 6.3.1 : architecture générale de l'outil de visualisation

Pour réaliser la solution exposée brièvement au chapitre précédent, notre idée est de permettre d'ajouter un texte de description permettant de décrire :

1) les objets modélisés dans l'arbre abstrait que l'utilisateur désire voir représentés graphiquement à l'écran, ainsi que la représentation graphique souhaitée pour ces objets,
2) les relations qui lient ces objets ainsi que la représentation graphique souhaitée pour ces relations.

Cette description va être "compilée" pour donner la table de description des objets et des relations. Cette table, ainsi que la table des concepts graphiques et la table des correspondances entre structures logiques et physiques vont diriger la décompilation graphique générique.

A partir d'une commande de l'utilisateur saisie par le module de saisie, ce dernier appellera le module de décompilation graphique qui déclenchera toute la "machinerie" nécessaire à l'affichage des objets et des relations précisées par l'utilisateur à l'écran. La "machinerie" est la suivante :

- le module de décompilation graphique, à partir de la table de description des objets et des relations et des autres tables (table des concepts graphiques et table des correspondances entre structures logiques et physiques, cfr 6.9) créera (à la demande de l'utilisateur) la structure intermédiaire graphique sans notion de positions qui contiendra toutes les informations nécessaires pour l'affichage,

- le module d'annotation de la structure intermédiaire graphique sans notion de positions annotera cette structure en fonction de la taille de la fenêtre dans laquelle cette structure doit être affichée. L'annotation consiste à attribuer à chaque noeud (représentant un objet ou une relation) de la structure intermédiaire graphique une position qui sera celle à laquelle l'information contenue dans ce noeud sera affichée à l'écran,

- le module d'affichage qui se contentera d'afficher la structure intermédiaire graphique avec notion de positions dans la fenêtre.

Cette architecture est présentée à la figure 6.3.1, partie inférieure.
Les sous-problèmes identifiés sont les suivants :

1) pour le module de décompilation graphique

   - définition d'un langage permettant de préciser quels sont les objets et les relations entre ces objets que l'utilisateur désire représenter graphiquement à l'écran. Il s'agit de définir un **langage de description d'objets et de relations inter-objets,**

   - définition d'un langage permettant de préciser quelle est la représentation graphique souhaitée pour les différents objets et relations considérés ; il s'agit d'un **langage de description graphique.** Ces deux langages sont en fait intégrables en un seul. Nous avons distingué deux langages pour faire la séparation entre :

- ce qu'il faut représenter, le QUOI, décrit au moyen du L.D.O.R.,
- comment il faut le représenter, le COMMENT, décrit au moyen du langage de description graphique.

L'intégration de ces deux langages constitue un méta-langage permettant de définir un langage d'interface graphique (L.I.G.),

2) déterminer le contenu de la **table de description des objets et des relations**. Cette table assure la généricité de l'outil par rapport à un langage d'interface graphique ; à partir du langage de description d'objets et de relations inter-objets, il est possible de définir la partie sémantique, syntaxique et lexicale d'un langage d'interface graphique ; cette table est obtenue par "compilation" d'un texte de description écrit grâce au langage de description d'objets et de relations inter-objets et au langage graphique,

3) déterminer le contenu de la **table des correspondances entre structures logiques et physiques** qui permet d'assurer la généricité de l'outil par rapport à une structure physique d'arbre abstrait. Déterminer le contenu de la **table des concepts graphiques** qui permet d'assurer la généricité de l'outil par rapport aux concepts graphiques (RECTANGLE, CERCLE ...),

4) définir la structure physique de la **structure intermédiaire graphique**, c'est-à-dire déterminer les informations qui y sont nécessaires et leur mode d'agencement,

5) définir le mode d'**annotation** de la structure intermédiaire graphique, ce qui pose des problèmes (cfr section 6.11) dus à la nature graphique/textuelle des informations contenues dans les noeuds de la structure intermédiaire graphique,

6) définir le mode d'**affichage** de la structure intermédiaire graphique.

Nous allons passer en revue dans les sections suivantes les différents composants de l'architecture et les sous-problèmes identifiés qui y correspondent.

6.4 Syntaxe abstraite du langage SACSO

Cette section nous décrit brièvement la structure générale de tout arbre abstrait représentant une spécification SACSO. Nous utiliserons cette structure générale dans la suite pour nos exemples sur le langage de description d'objets et de relations inter-objets. Pour rappel, une spécification est composée d'un énoncé (définition des opérations) et d'un univers (définition des types). Une spécification utilise une bibliothèque (environnement), c'est-à-dire un ensemble de spécifications. Une spécification est représentée sous la forme d'un arbre. La structure de celui-ci décrit le mode de dérivation de la spécification selon les règles de production définissant la grammaire du langage ; à chaque non terminal du langage correspond un type de noeud et la structure de ce type de noeud correspond au membre droit de la règle grammaticale correspondante.

Les figures 6.4.1 à 6.4.7 décrivent des structures de sous-arbres abstraits qui seront nécessaires à la compréhension de l'exemple de la section 6.5.1 ; nous utilisons le formalisme graphique des constructeurs de types présentés à la section 5.2.

Un arbre abstrait décrit comment une spécification est représentée de façon interne, et stockée.
Pour afficher une spécification à l'écran, il faut ajouter du sucre syntaxique à l'information présente dans son arbre abstrait.

Structure d'un noeud **SPECIF**

Une spécification est composée de :

- la spécification de l'énoncé,
- la spécification de l'univers,
- une description.

En plus, s'ajoutent un nom ainsi qu'une liste de spécifications utilisées. En effet, un problème à résoudre se situe souvent dans un environnement plus vaste, qui a peut-être déjà fait l'objet d'une spécification pouvant être réutilisée. Au minimum, toutes les spécifications utiliseront au moins certaines spécifications, celles des types de base et de leurs opérations associées prédéfinis dans la bibliothèque (cfr figure 6.4.1).

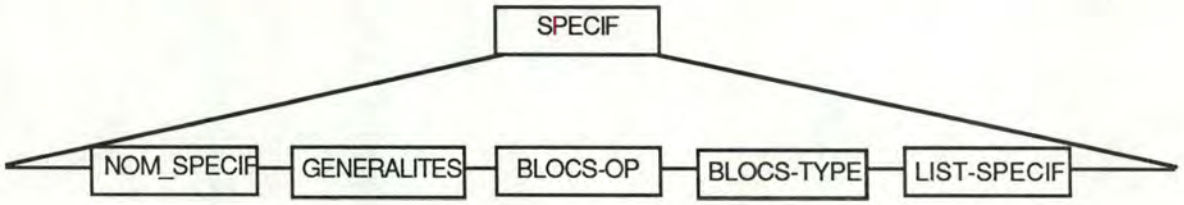


Figure 6.4.1 : structure d'un noeud SPECIF

Noeud SPECIF -> PC [nom : NOM-SPECIF,
généralités : GENERALITES,
énoncé : BLOCS-OP,
univers : BLOCS-TYPE,
spécifs : LIST-SPECIF]

Structure d'un noeud **BLOCS-TYPE**

Une spécification d'univers est constituée de la spécification de chaque type et sera représentée par une table, ce qui permet d'accéder à la description d'un type donné s'il est défini (cfr figure 6.4.2).
Type BLOCS-TYPE -> T[nom, BLOC-TYPE]

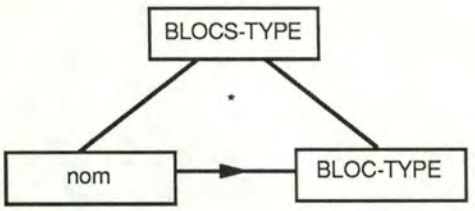


Figure 6.4.2 : structure d'un noeud BLOCS-TYPE

Strcuture d'un noeud **BLOC-TYPE**

Une description de type est constituée des éléments suivants :

- le nom du type
- un lexique
- une liste d'arguments, pour les types paramétrés
- une liste d'opérations associées
- sa structure
- un invariant

A ceci ont été ajoutés une liste d'opérations et de types référencés (cfr figure 6.4.3).

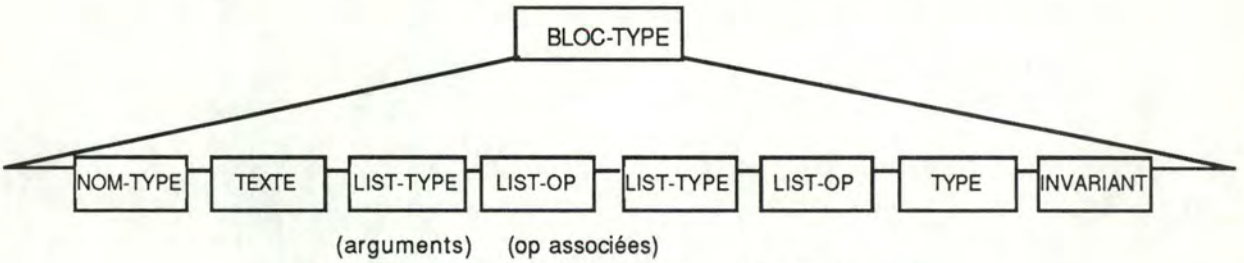


Figure 6.4.3 : structure d'un noeud BLOC-TYPE

Type BLOC-TYPE -> PC[nom : NOM-TYPE,
lex : TEXTE,
arg : LIST-TYPE,
op-assoc : LIST-OP,
type : LIST-TYPE,
op : LIST-OP,
def : TYPE,
invariant : INVARIANT]

Sructure d'un noeud **TYPE**

Le type d'un objet peut être défini (une instanciation d'un type paramétré, ou un type déjà défini), ou indéfini (cfr figure 6.4.4).

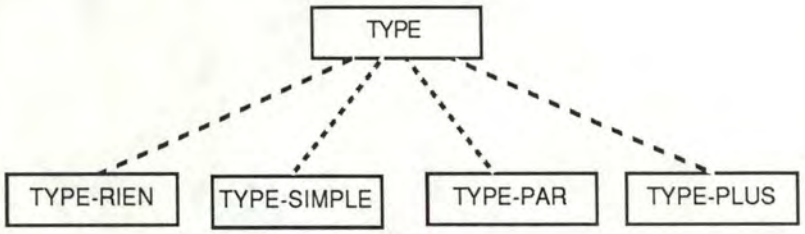


Figure 6.4.4 : structrure d'un noeud TYPE

Type TYPE -> U [TYPE-RIEN, TYPE-SIMPLE, TYPE-PAR, TYPE-PLUS]

Structure d'un noeud **TYPE-PLUS**

Un noeud de ce type représente un type un peu particulier. Ce type remplace une liste de paramètres formels.

Structure d'un noeud **TYPE-RIEN**

Un noeud de ce type représente un type indéfini.

Structure d'un noeud **TYPE-SIMPLE**

Un noeud de ce type représente un type simple, c'est-à-dire un type qui n'est pas paramétré.

Structure d'un noeud **TYPE-PAR**

Un noeud de ce type représente un type paramétré. Il est composé du nom du type paramétré, et de la liste des paramètres (liste de noeud TYPE). Dans le cas d'un type produit cartésien, il faut ajouter la liste des opérations d'accès aux champs (cfr figure 6.4.5).

Type TYPE-PAR -> U[PC [nom : MAJUSCULE, arg : TYPES]
PRODUIT-CARTESIEN]

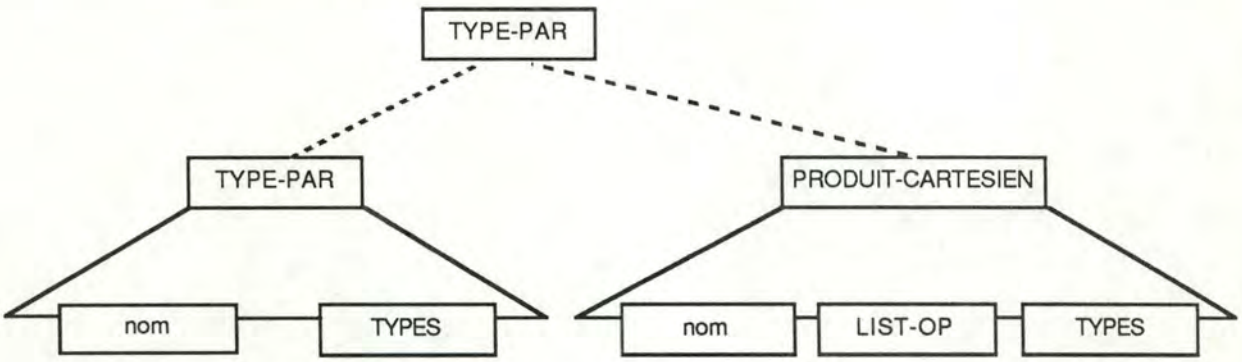


Figure 6.4.5 : structure d'un noeud TYPE-PAR

Structure d'un noeud **PRODUIT-CARTESIEN**

Un noeud de ce type représente le type paramétré produit cartésien (cfr figure 6.4.5).

Type PRODUIT-CARTESIEN -> PC[nom : MAJUSCULE,
op : LIST-OP
arg : TYPES]

Structure d'un noeud **TYPES**

Un noeud de ce type représente une liste de noeuds de type TYPE (cfr figure 6.4.6).

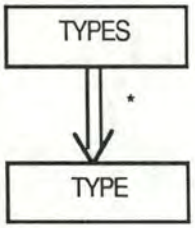


Figure 6.4.6 : structure d'un noeud TYPES

Type TYPES -> S[TYPE]

Structure d'un noeud **LIST-TYPE**

Un noeud de ce type représente une liste de noms de type (cfr figure 6.4.7).

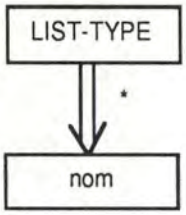


Figure 6.4.7 : structure d'un noeud LIST-TYPE

Type LIST-TYPE -> S[MAJUSCULE]

6.5 Le langage de description d'objets et de relations inter-objets

6.5.1 Présentation du langage

L'objectif de ce langage est de fournir un moyen de décrire les types d'objets et les relations entre ces types d'objets qui devront être représentés graphiquement. Ce langage est en fait un méta-langage permettant de définir un langage d'interface graphique particulier en définissant la partie syntaxique, lexicale et sémantique de ce dernier. L'hypothèse de départ est qu'il ne sera possible de représenter que des types d'objets et des relations "calculables" qui sont modélisées dans un arbre abstrait. Le terme "calculable" est utilisé pour désigner les relations dont les objets qu'elle relie peuvent être extraits au moyen d'un parcours dans un arbre abstrait.

Une première idée qui vient à l'esprit est d'opérer de la même manière que lors du passage de la syntaxe abstraite à la syntaxe concrète de sortie dans les éditeurs syntaxiques. Il faut

décorer l'arbre abstrait avec de la syntaxe concrète (sucre syntaxique). Seulement dans notre cas il ne s'agit pas de décorer la syntaxe abstraite uniquement avec du sucre syntaxique de type textuel. Ce qu'il faut faire c'est décorer la syntaxe avec des concepts graphiques. Le formalisme utilisé dans la partie gauche de la flèche de l'exemple suivant est celui de METAL [Mélèse 82], et du METAL "étendu" aux concepts graphiques dans la partie droite.

Exemple 6.5.1:

```
if (*cond, *stat1, *stat2)  -> [RECTANGLE (*cond)
                                RECTANGLE ("then" *stat1)
                                RECTANGLE ("else" *stat2)]
```

A gauche de la flèche il y a un schéma d'arbre abstrait et à droite un format de décompilation graphique du schéma de l'arbre abstrait.

Ce formalisme signifie que toute occurrence du schéma d'arbre abstrait à gauche de la flèche sera représenté graphiquement par

- un rectangle contenant la partie condition (contenu de *cond) de l'occurrence du schéma d'arbre abstrait,
- d'un rectangle contenant la chaîne de caractères "then" suivie de la partie then (contenu de *stat1) de l'occurrence du schéma d'arbre abstrait,
- d'un rectangle contenant la chaîne de caractères "else" suivie de la partie else (contenu de *stat2) de l'occurrence du schéma d'arbre abstrait.

On voit clairement qu'il manque ici des attributs supplémentaires dans la partie droite de la flèche pour préciser la composition spatiale souhaitée pour ces rectangles. Supposons que les rectangles soient alignés. Dans la suite, nous allons oublier ce problème de composition spatiale qui sera réintroduit par la suite. Notons, que ce problème de préciser la composition spatiale souhaitée, peut être résolu du moins dans le cas de décompilation textuelle, par des attributs de composition (h, v, ...) comme dans PPML [Brossard 86]. PPML (**P**retty**P**rinting **M**eta**L**anguage) est un langage qui permet de construire des décompilateurs textuels grâce à un formalisme proche de celui présenté à l'exemple 6.5.1.
Le problème revient donc dans ce cas à définir un langage de décompilation graphique.
Comment exprimer dans ce langage des relations entre types d'objets ?

Il faut introduire une notion de lien graphique (qui représente de manière graphique une relation existant entre deux objets ou concepts représentés eux aussi graphiquement) ; un lien graphique est défini comme une liaison graphique (droite, arc, flèche ...) entre deux éléments graphiques (rectangle, cercle ...).

Par exemple on veut qu'une conditionelle if soit représentée de manière graphique. Une présentation graphique possible de la conditionnelle :

```
if toto = 5 then trouve := true
            else trouve := false
```

pourrait être celle donnée à la figure 6.5.1.

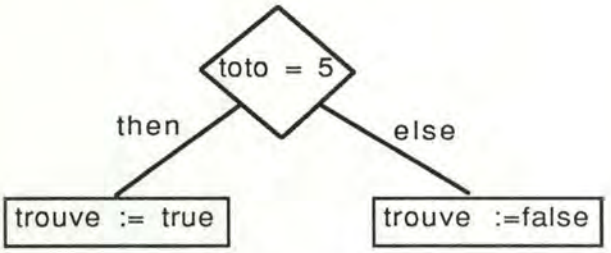


Figure 6.5.1 : représentation graphique possible de la conditionnelle

Cela pourrait se faire de la manière suivante :

if (*cond, *stat1, *stat2) -> [LOSANGE (*cond)
lié par DROITE ("then") à RECTANGLE (*stat1)
lié par DROITE ("else") à RECTANGLE (*stat2)]

Ceci est rendu possible grâce au concept "lié par à" du langage.

Le principe de décorer la syntaxe abstraite avec des concepts graphiques est envisageable si la relation (sémantique) représentée par les liaisons graphiques à l'écran est modélisée de manière arborescente dans la syntaxe abstraite. En d'autres termes, il faut pouvoir représenter de manière graphique des relations qui existent entre des objets modélisés dans l'arbre abstrait. Mais dans certains cas, pour passer d'un objet relié à un autre objet un parcours complexe de l'arbre est nécessaire. Cela signifie donc qu'il s'agit de relations qui doivent être calculées. La solution de décorer la syntaxe abstraite comme pratiquée ci-dessus présente donc une limite assez importante. Il faut adjoindre un moyen de décrire la relation qui doit être représentée.

La solution proposée consiste à écrire un **texte de description** qui contient la représentation des objets et relations à représenter. Dans ce langage de description d'objets et de relations inter-objets, un arbre abstrait est vu comme une occurrence du type abstrait "ARBRE ABSTRAIT". Ce type abstrait composé de types de noeuds est construit à partir des constructeurs de types SACSO. Dès lors, le parcours dans un arbre abstrait ainsi que l'accès à l'information contenue dans cet arbre sont exprimés en termes d'opérations sur des types abstraits. Ce sont les constructeurs de types et les opérations SACSO (voir annexe 3) qui sont utilisés pour décrire la structure des types abstraits. Dans le cas d'un éditeur graphique guidé par la syntaxe où il est nécessaire de modifier un arbre abstrait, cette vue d'un arbre abstrait comme une occurrence d'un type abstrait est particulièrement intéressante. Toutes les opérations de modification d'un arbre abstrait peuvent être exprimées en termes d'opérations sur des types abstraits.

Les exemples utilisés pour illustrer les différentes sections sont basés sur la structure générale de tout arbre abstrait représentant une spécification SACSO (cfr section 6.4). Pour la syntaxe précise du L.D.O.R., on consultera l'annexe 4.

- **section OBJETS** : cette section contient les types d'objets (types de noeuds de l'arbre abstrait) pour lesquels une représentation graphique à l'écran est demandée ainsi que la manière dont un objet de ce type doit être représenté. La section OBJETS définit un filtrage, sur l'arbre, des noeuds qui doivent être retenus pour être représentés.
La syntaxe utilisée est la suivante :

TYPE-DE-NOEUD représenté par CONCEPT-GRAPHIQUE

Elle signifie que toute occurrence du type de noeud TYPE-DE-NOEUD doit être représentée à l'écran par le concept graphique CONCEPT-GRAPHIQUE. La syntaxe utilisée pour CONCEPT-GRAPHIQUE est présentée dans la section 6.6 sur le langage de description graphique.

Exemple :
Section OBJETS
BLOC-TYPE représenté par
RECTANGLE (accès [BLOC-TYPE, NOM-TYPE])

Cette section définit que toute occurrence du type de noeud BLOC-TYPE doit être représentée graphiquement par un rectangle comprenant le contenu du champ NOM-TYPE de cette occurrence du type de noeud BLOC-TYPE.

- **section RELATIONS** : cette section décrit les relations entre les types d'objets dont les occurrences doivent être représentées à l'écran. En fait, cette section sert à "déclarer" les relations. Une relation peut être définie sur plusieurs types de noeuds.

Exemple :
Supposons que nous veuillons déclarer la relation UTILISE qui relie des objets de type BLOC-TYPE à des objets de type BLOC-REL ; les objets de type BLOC-REL sont eux-mêmes reliés à des objets de type BLOC-TYPE. La relation UTILISE doit donc être déclarée sur les types de noeuds BLOC-TYPE et BLOC-REL. Cependant, la manière d'obtenir tous les noeuds de type BLOC-TYPE reliés par la relation UTILISE à un noeud de type BLOC-REL est différente de la manière d'obtenir tous les noeuds de type BLOC-REL reliés par la relation UTILISE à un noeud de type BLOC-TYPE. Pour cette raison, nous introduisons la notion de sous-relation. Une relation peut être décomposée en plusieurs sous-relations, chacune pouvant être définie dans la section DEFINITIONS_RELATIONS et permettant ainsi de déclarer des relations définies sur plusieurs types de noeuds. Une sous-relation ne peut être définie que sur deux types de noeuds.
Dès lors, la relation UTILISE sera décomposée en deux sous-relations UTILISE1 et UTILISE2. La section RELATIONS se présentera donc comme suit :

relation utilise
BLOC-TYPE : niv i (utilise1) BLOC-REL : niv i+1
(utilise2) BLOC-TYPE : niv i+2

Dans la suite, nous utilisons l'expression "niveau i de la relation" pour désigner l'ensemble des occurrences d'un type de noeud obtenu par application de la définiton d'une sous-relation de la relation (cfr section DEFINITIONS_RELATIONS) à l'ensemble des occurrences d'un type de noeud de niveau supérieur. Les niveaux sont définis comme suit :

- le niveau 0 est composé d'une occurrence d'un type de noeud,
- le niveau i est composé de l'ensemble des occurrences d'un type de noeud obtenu en appliquant la relation à l'ensemble des occurrences d'un type de noeud de niveau i-1.

La syntaxe utilisée est la suivante :

relation NOM-RELATION
TYPE-DE-NOEUD1 : niv i (NOM-SOUS-RELATION)
TYPE-DE-NOEUD2 : niv j

Elle signifie que la relation de nom NOM-RELATION, est composée de la sous-relation de nom NOM-SOUS-RELATION qui relie le type de noeud TYPE-DE-NOEUD1 du niveau i de la relation de nom NOM-RELATION au type de noeud TYPE-DE-NOEUD2 du niveau j de la relation de nom NOM-RELATION.

- **section DEFINITIONS_RELATIONS** : cette section contient la définition des relations présentées dans la section RELATIONS. Une relation étant composée de sous-relation(s), cette section contient en fait les définitions des différentes sous-relations composant les relations déclarées dans la section RELATIONS. Cette section décrit le parcours a effectuer dans l'arbre abstrait à partir d'un noeud pour obtenir les noeuds liés à ce noeud par la relation qui est en cours de définition. Ce parcours s'exprime en termes d'opérations prédéfinies sur les constructeurs de types SACSO puisque les types de noeuds du type abstrait "ARBRE ABSTRAIT" sont construits à partir de ces constructeurs.

Exemple :
Section DEFINITIONS_RELATIONS

relation utilise définie_par
nom-type := accès [BLOC-TYPE : niv i, NOM-TYPE]
table_type := accès [noeud_racine, BLOCS-TYPE]
bloc_type := acct [table_type, nom_type]
list_type := accès [bloc_type, LIST-TYPE]
iter pour j de 1 à taille(list_type)
BLOC-TYPE : niv i+1 obtenu_par acct [table_type, j]

La signification de cette définition de relation est la suivante. Recevant un noeud de type BLOC-TYPE (BLOC-TYPE : niv i) représentant un type SACSO, on accède au champ NOM-TYPE de ce noeud pour obtenir le nom de ce type SACSO. Ensuite, on accède au champ BLOCS-TYPE du noeud racine de l'occurrence du type abstrait "ARBRE ABSTRAIT" considérée. On obtient un noeud de type BLOCS-TYPE qui est une table contenant tous les noms des types utilisés dans la spécification SACSO représentée par l'occurrence du type abstrait "ARBRE ABSTRAIT" (cfr 6.4). Ensuite, on accède à cette table avec nom-type pour obtenir le noeud de type BLOC-TYPE représentant dans l'occurrence du type abstrait le type SACSO de nom nom-type. Ayant ce noeud, on accède à son champ LIST-TYPE qui contient tous les noms de types utilisés par le type SACSO de nom nom-type. Tout ce processus est réappliqué récursivement à chaque élément de cette liste. On obtient ainsi récursivement l'arbre des types d'un type.

- **section COMPOSITION** : cette section décrit la syntaxe du langage d'interface graphique, c'est-à-dire le mode de composition spatiale des représentations graphiques des occurrences des types d'objets (types de noeuds) donnés dans la section OBJETS. Les modes de composition spatiale sont basés sur les points cardinaux et sont les suivants :

  - N : nord
  - E : est
  - S : sud
  - O : ouest
  - N-O : nord-ouest
  - N-E : nord-est
  - S-E : sud-est
  - S-O : sud-ouest
  - I : intérieur

  Le mode de composition spatiale est exprimé en utilisant la syntaxe suivante :

  TYPE-NOEUD1 : niv k [MODE-DE-COMPOSITION.NOMBRE]
  TYPE-NOEUD2 : niv h

  c'est-à-dire en termes de types de noeuds et de niveaux de la relation.
  La signification est la suivante :

  toute occurrence du type de noeud TYPE-NOEUD2 du niveau h de la relation doit être placée au(à) MODE-DE-COMPOSITION (valeur N, E, S, ...) de toute occurrence du type de noeud TYPE-NOEUD1 du niveau k de la relation.
  La syntaxe MODE-DE-COMPOSITION.NOMBRE est utilisée pour désigner une occurrence du mode de composition spatiale MODE-DE-COMPOSITION et servira dans la section REPRESENTATIONS_RELATIONS à préciser la représentation graphique souhaitée pour cette occurrence particulière du mode de composition. NOMBRE est un entier identifiant l'occurrence du mode de composition spatiale

MODE-DE-COMPOSITION parmi l'ensemble des occurrences de ce mode de composition spatiale.

Exemple :

BLOC-OP : niv i [O.1] BLOC-OP : niv i+1

signifie que toute occurrence du type de noeud BLOC-OP du niveau i+1 de la relation doit être représentée (à l'écran) à l'ouest de toute occurrence du type de noeud BLOC-OP du niveau i de la relation.

De plus il est possible de pouvoir préciser le mode de composition d'occurrences spécifiques de types de noeuds. Cela permet d'affiner la définition de la représentation souhaitée.
La syntaxe utilisée est la suivante :

TYPE-NOEUD1 : niv k occ j [MODE-DE-COMPOSITION.NOMBRE]
TYPE-NOEUD2 : niv h occ m

Elle signifie que l'occurrence m du type de noeud TYPE-NOEUD2 du niveau h de la relation doit être positionnée au(à) MODE-DE-COMPOSITION (valeur : N, O, S ...) de l'occurrence j du type de noeud TYPE-NOEUD1 du niveau k de la relation.
Pour la définition du mode de composition, le problème consiste à donner une définition récursive du mode de composition souhaité.
Puisque la composition est toujours exprimée en termes de niveaux de relation une solution pour définir le mode de composition souhaité consiste à donner :

- le mode de composition souhaité pour l'occurrence du type de noeud de départ, à savoir le mode de composition souhaité entre les occurrences du type de noeud du niveau 1 de la relation, et l'occurrence du type de noeud du niveau 0 de la relation,
- le mode de composition souhaité pour le cas général, c'est-à-dire entre les occurrences du type de noeud du niveau n de la relation et les occurrences du type de noeud du niveau n-1 de la relation,
- le mode de composition souhaité pour les exceptions au cas général.

Cela donnerait par exemple :

Section COMPOSITION

Cas départ

règle BLOC-TYPE : niv 0 [S.1] BLOC-TYPE : niv 1
règle BLOC-TYPE : niv 1 occ 0 [E.1] BLOC-TYPE : niv 1 occ 1

Cas général

règle BLOC-TYPE : niv n-1 [S.1] BLOC-TYPE : niv n
règle BLOC-TYPE : niv n occ i [E.1] BLOC-TYPE : niv n occ i+1

Cas exception

règle BLOC-TYPE : niv 3 [E.1] BLOC-TYPE : niv 2

Les occurrences de modes de composition représentent en fait une relation sémantique entre deux objets qui est représentée spatialement. Il est possible dans la section REPRESENTATIONS_RELATIONS de préciser la représentation graphique souhaitée pour cette relation.

- **section REPRESENTATIONS_RELATIONS** : cette section reprend toutes les occurrences de modes de composition définies à la section COMPOSITION et précise la représentation graphique souhaitée.
La syntaxe est la suivante :

MODE-DE-COMPOSITION.NOMBRE représenté par
CONCEPT-GRAPHIQUE
ou RIEN
ou COLLE

Elle signifie que l'occurrence du mode de composition MODE-DE-COMPOSITION doit être représentée à l'écran par le concept graphique CONCEPT-GRAPHIQUE. La syntaxe à utiliser pour CONCEPT-GRAPHIQUE est présentée à la section 6.6. L'utilisation du mot clé RIEN permet de dire que les occurrences des types de noeuds représentées ne doivent pas être collées mais qu'il n'y a pas de lien graphique (droite, flèche, ...) entre ces occurrences. Par contre, l'utilisation de COLLE signifie que les représentations des occurrences des types de noeuds doivent être collées.

Exemple :

Reprenons un exemple plus complet pour expliquer l'objet de cette section. Soit la partie de texte de description suivante :

DESCRIPTION EXEMPLE

Section OBJETS

BLOC-TYPE représenté par
RECTANGLE (accès [BLOC-TYPE, NOM-TYPE])

Section RELATIONS

relation utilise_type
BLOC-TYPE : niv i (utilise) BLOC-TYPE : niv i+1

Section DEFINITIONS_RELATIONS

sans intérêt

Section COMPOSITION

Cas départ
règle BLOC-TYPE : niv 0 [S.1] BLOC-TYPE : niv 1
règle BLOC-TYPE : niv 1 occ 0 [E.1] BLOC-TYPE : niv 1 occ 1

Cas général
règle BLOC-TYPE : niv n-1 [S.1] BLOC-TYPE : niv n
règle BLOC-TYPE : niv n occ i [E.1] BLOC-TYPE : niv n occ i+1

Cas exception
règle BLOC-TYPE : niv 3 [S.2] BLOC-TYPE : niv 2

Section REPRESENTATIONS_RELATIONS

S.1 représenté par DROITE
O.1 représenté par RIEN
S.2 représenté par COLLE

La section REPRESENTATIONS_RELATIONS définit que
- l'occurrence S.1 du type de mode de composition spatiale S doit être représentée par une droite,
- l'occurrence O.1 du type de mode de composition spatiale O doit être représentée par rien,
- l'occurrence S.2 du type de mode de composition spatiale S doit être représentée par COLLE.

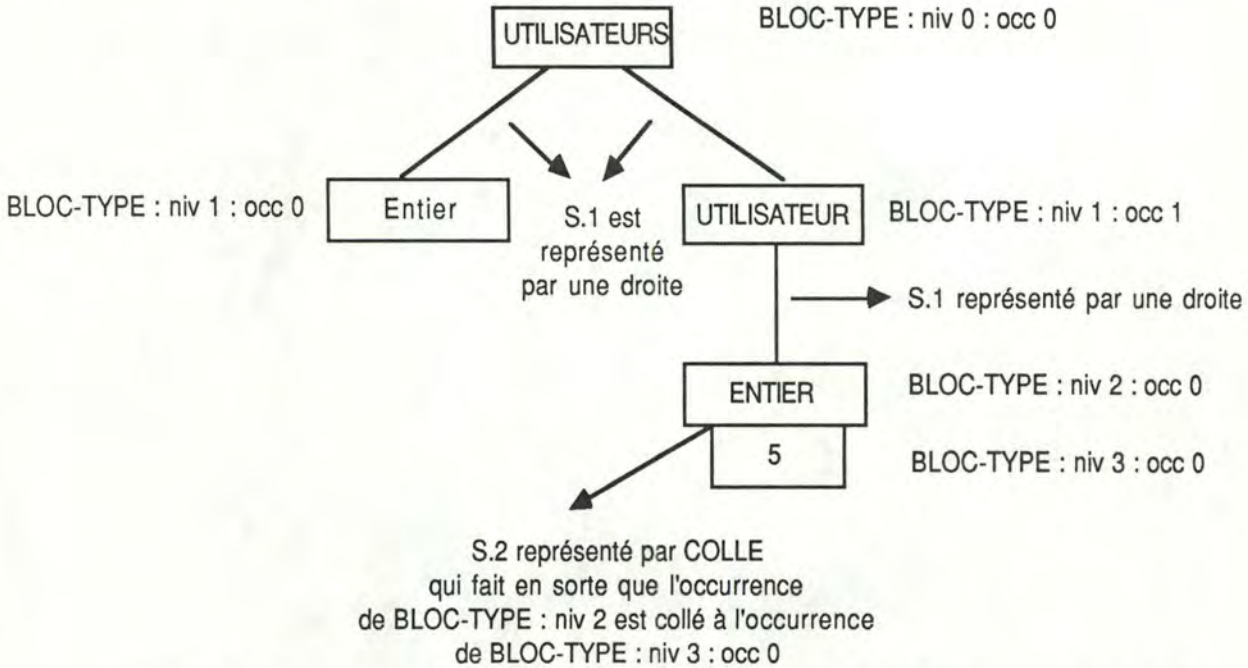


Figure 6.5.2 : exemple de représentation graphique respectant la syntaxe graphique définie dans la section COMPOSITION

Une représentation graphique, parmi d'autres, respectant la description donnée dans les section COMPOSITION et section REPRESENTATIONS_RELATIONS est présentée à la figure 6.5.2.

- notion de **CONTRAINTE** : une contrainte est associée à un type d'objet. Une contrainte est composée d'une suite d'opérations (corps). Celles-ci doivent être réalisées chaque fois qu'une occurrence du type de noeud à laquelle cette contrainte est associée est rencontrée lors du parcours d'une occurrence du type abstrait "ARBRE ABSTRAIT". Cette contrainte doit être appliquée chaque fois qu'un noeud du type spécifié est rencontré. Une contrainte peut être utilisée dans n'importe quelle section. La syntaxe utilisée est la suivante :

  contrainte NOM-CONTRAINTE (TYPE-DE-NOEUD)
  début
  < corps de la contrainte >
  fin

  Dans le cas où la construction d'un **éditeur graphique dirigé par la syntaxe** est envisagée, il faudrait ajouter une section ACTION supplémentaire définie comme suit.

- **section ACTION** : cette section décrit les actions qui sont permises sur la représentation graphique à l'écran. Chaque opération permise sur la représentation graphique serait liée à une procédure de modification de l'arbre abstrait.
  Les actions devraient exprimer les actions permises sur la représentation graphique. Il y a deux types d'actions :

1) des actions menant à une modification de l'arbre abstrait. Il faut donc spécifier ces actions, la manière de les détecter, et la procédure de modification de l'arbre abstrait qui doit être exécutée en cas de survenance d'une telle action,

   Exemple :
   Section ACTIONS
   (événement = MiddleButtonPressed)
   et (est-type [noeud_pointé, BLOC-TYPE])
   --> nouveau_nom_type := SAISIE_NOM ()
   table_type := accès [noeud_racine, BLOCS-TYPE]
   anc_nom_type := accès [noeud_pointé, NOM-TYPE]
   bloc_type := acct [table_type, ancien_nom_type]
   table_type := mod [supt [table_type, anc_nom_type],
   nouveau_nom_type,
   noeud_pointé]

   Cette section spécifie que si l'utilisateur clique avec le bouton du milieu de la souris dans la représentation graphique d'une occurrence du type de noeud BLOC-TYPE (représentant un type), il faut :
   - saisir le nouveau nom que l'utilisateur veut donner à ce type,

- modifer le nom de ce type dans la table des types BLOCS-TYPE qui reprend le nom de tous les types utilisés dans une spécification.

2) des actions menant à une consultation de l'arbre abstrait, ou à l'appel d'un outil consultant l'arbre abstrait. Dans ce cas, l'action doit être spécifiée comme précédemment.
Il faudrait de plus une section **Propriétés** qui permettrait d'exprimer quelles sont les propriétés qui doivent être vérifiées pour la construction ou modification d'un arbre abstrait.

Exemple :

Section PROPRIETES

appt [table_type, nouveau_nom_type] = faux

Cette "propriété" exprime que le nouveau nom (nouveau_nom_type) donné au type SACSO par l'utilisateur ne doit pas déjà avoir été défini dans la spécification ; ce nom ne doit pas se trouver dans la table table_type.

Dans notre cas, afin de cacher la structure interne de la structure intermédiaire graphique, il faut au moins une procédure qui reçoive en entrée la coordonnée (position en x et en y) à laquelle l'utilisateur a cliqué dans la structure intermédiaire graphique de manière à lui fournir le noeud de l'arbre abstrait vers lequel pointe le noeud de la structure intermédiaire auquel correspond cette position. Toute transformation de l'arbre abstrait doit être répercutée sur la représentation graphique quel que soit l'arbre abstrait.

Pour un **éditeur graphique** dirigé par la syntaxe, en plus de l'opération de pointage il y a des opérations de modification de l'arbre abstrait suite à une modification de la représentation graphique à l'écran. Ces opérations devront être exprimées en termes de modification et création de noeuds dans l'arbre abstrait. La structure intermédiaire sera automatiquement mise à jour, en fonction des modifications de l'arbre abstrait.

Le texte de description utilisant le langage de description des objets et des relations tel que défini prédédemment serait le suivant pour représenter l'arbre des types d'un type d'une spécification SACSO. Nous supposons que l'utilisateur pointera sur le type pour lequel il veut obtenir une représentation graphique de son arbre des types et que le noeud racine noeud_racine de l'arbre abstrait représentant la spécification SACSO est connu. Ce type qui est représenté par un noeud de type BLOC-TYPE dans l'arbre abstrait constituera le noeud de départ dans l'arbre abstrait pour le module de recherche et de création de la structure intermédiaire sans notion de positions. Dans le texte de description présenté dans la suite, ce noeud de départ de type BLOC-TYPE est en fait la seule occurrence de BLOC-TYPE : niv 0. Le texte de description se présente comme suit :

DESCRIPTION TYPE_SACSO

Section OBJETS

BLOC-TYPE représenté_par RECTANGLE (accès [BLOC-TYPE, NOM-TYPE])

Section RELATIONS

relation utilise_type
BLOC-TYPE : niv i (utilise) BLOC-TYPE : niv i+1

Section DEFINITIONS_RELATIONS

relation utilise définie_par
nom-type := accès [BLOC-TYPE : niv i, NOM-TYPE]
table_type := accès [noeud_racine, BLOCS-TYPE]
bloc_type := acct [table_type, nom_type]
list_type := accès [bloc_type, LIST-TYPE]
iter pour j de 1 à taille(list_type)
BLOC-TYPE : niv i+1 obtenu_par acct [table_type, j]

Section COMPOSITION

Cas départ
règle BLOC-TYPE : niv 0 [S.1] BLOC-TYPE : niv 1
règle BLOC-TYPE : niv 1 occ 0 [E.1] BLOC-TYPE : niv 1 occ 1

Cas général
règle BLOC-TYPE : niv n-1 [S.1] BLOC-TYPE : niv n
règle BLOC-TYPE : niv n occ m-1 [E.1] BLOC-TYPE : niv n occ m

Section REPRESENTATIONS_RELATIONS

contrainte représentation (BLOC-TYPE)
début
type_noeud := accès [BLOC-TYPE, TYPE]
si (est-type [type_noeud, PRODUIT-CARTESIEN])
alors
E.1 représenté_par DROITE
si j = 1 ou j = taille (list_type)
alors
S.1 représenté_par DROITE_DROITE
sinon
S.1 représenté_par RIEN
sinon
si et ((est-type [type_noeud, TYPE-PAR]),
(eq-chaine (accès [type_noeud, NOM], "S"))
alors
S.1 représenté_par DOUBLE_FLECHE_BAS ("*")
O.1 représenté_par RIEN

```
si et ((est-type [type_noeud, TYPE-PAR]),
        (eq-chaine (accès [type_noeud, NOM], "E"))
alors
        S.1 représenté_par DROITE ("*")
        E.1 représenté_par RIEN

si et ((est-type [type_noeud, TYPE-PAR]),
        (eq-chaine (accès [type_noeud, NOM], "U"))
alors
        S.1 représenté_par DROITE_POINTILLEE
        E.1 représenté_par RIEN

si et ((est-type [type_noeud, TYPE-PAR]),
        (eq-chaine (accès [type_noeud, NOM], "T"))
alors
        S.1 représenté_par DROITE_ARC
        E.1 représenté_par DROITE_FLECHE
fin
```

La section **DEFINITIONS_RELATIONS** définit le parcours qui doit être effectué dans l'arbre abstrait pour obtenir les différents objets à représenter graphiquement. Dans un arbre abstrait représentant une spécification SACSO, tout type de cette spécification est représenté par un noeud de type BLOC-TYPE. Recevant le noeud de départ, de type BLOC-TYPE, les noms de ses noeuds fils de type BLOC-TYPE (puisque utilise a été définie comme une relation d'un BLOC-TYPE vers un autre) se trouvent dans la liste LIST-TYPE qui est un champ du noeud BLOC-TYPE (cfr description de la syntaxe abstraite de SACSO en 6.4). Il s'agit d'appliquer la relation récursivement à tous les éléments de la liste LIST-TYPE en trouvant à quel BLOC-TYPE correspondent les noms en accédant à la table BLOCS-TYPE avec leur nom.

La section **REPRESENTATIONS_RELATIONS** permet de définir une représentation différente de la relation utilise suivant le constructeur de type avec lequel a été construit le type. Pour chaque noeud de type BLOC-TYPE on accède au champ TYPE de ce noeud pour obtenir un noeud de type TYPE et on teste :

- si ce noeud est de type PRODUIT-CARTESIEN les occurrences de modes de compostion se voient attribuer les valeurs de concepts graphiques adéquates pour représenter un produit cartésien,
- si ce noeud est de type TYPE-PAR alors on teste le champ NOM du noeud pour connaitre le constructeur de type à partir duquel le type a été construit. En fonction du constructeur de types à partir duquel le type a été construit, les occurrences de mode de composition se voient attribuer les valeurs de concepts graphiques adéquates pour représenter le type.

6.5.2 Table de description des objets et des relations (T.D.O.R.)

Le(s) texte(s) de description écrit(s) en utilisant le langage de description des objets et des relations sera(ont) traité(s) et toute l'information nécessaire sera placée dans la T.D.O.R. La table sera utilisée par le module de recherche de l'information dans l'arbre abstrait.
Cette table contiendra les colonnes suivantes :

- **relations** (R) : cette colonne contient tous les noms des relations qui ont été déclarées dans les sections RELATIONS. Toutes les relations définies doivent avoir des noms uniques,

- **sous-relations** (S.R.) : cette colonne contient pour chaque relation tous les noms des sous-relations qui composent cette relation. Toutes les sous-relations déclarées doivent avoir des noms uniques

- **types de noeuds sources** (T.N.S.) : cette colonne contient pour chaque relation, les types de noeuds membres gauches des déclarations de sous-relations qui composent cette relation,

- **représentations graphiques des occurrences des TNS** (R.G.O.T.N.S.) : cette colonne contient pour chaque relation, la description de la représentation graphique souhaitée pour représenter toute occurrence du T.N.S. de chaque sous-relation qui compose cette relation,

- **types de noeuds cibles** (T.N.C.) : cette colonne contient pour chaque relation, les types de noeuds membres droits des déclarations de sous-relations qui composent cette relation,

- **représentations graphiques des occurrences des TNC** (R.G.O.T.N.C.) : cette colonne contient pour chaque relation, la description de la représentation graphique souhaitée pour représenter toute occurrence du T.N.C. de chaque sous-relation qui compose cette relation,

- **définitions des relations** (D.F.) : cette colonne contient pour chaque relation la définition de cette relation. Une relation étant composée de sous-relation(s), cette colonne reprend en fait pour chaque relation la définition des sous-relations qui la composent. La définition d'une sous-relation pouvant être complexe, deux alternatives sont envisageables :

  + soit un module transforme la définition de la sous-relation sous une forme exploitable avant de la mettre dans la table,
  + soit la définition est insérée comme telle dans la table et c'est le module de recherche et construction de la structure intermédiaire graphique qui la transforme pour ses propres besoins.

  La première alternative permet de modifier plus facilement l'outil de visualisation. En effet, en cas de changement de la syntaxe du langage de description d'objets et de

relations inter-objets, il suffit de modifer le module de transformation de manière à ce que l'information présente dans la table soit toujours représentée de la même manière. Il n'est pas nécessaire de modifier le module de recherche et construction de la structure intermédiaire.

- **composition** (C) : cette colonne contient pour chaque relation le mode de composition souhaité pour les occurrences de types de noeuds liées par chaque sous-relation qui compose cette relation. A nouveau cette description pouvant être complexe les deux alternatives présentées dans la colonne DF sont possibles,

- **représentations des occurrences des modes de compositions** (R.O.M.C.) : cette colonne contient pour chaque relation la représentation graphique souhaitée pour les occurrences de modes de compositions introduits dans la section COMPOSITION. La description pouvant être complexe les deux possibilités présentées dans la colonne D.F. peuvent être envisagées.

Il y aura un module chargé de la construction de la T.D.O.R. à partir du (des) texte(s) de description. Il suffira de construire cette T.D.O.R. une fois pour toute et la recréer chaque fois qu'un texte de description est modifié. La spécification par pré-post de ce module est la suivante.

Spécification pré-post du module de construction de la T.D.O.R.

construire_tdor (TEXTE_DESCR_GRAPHS, TDOR)

- argument :
  TEXTE_DESCR_GRAPHS : texte(s) de description graphique

- pré-condition :

- résultat
  TDOR : table de description des objets et des relations

- post-conditions :
  La TDOR sera construite à partir du(des) texte(s) de description TEXTE_DESCR_GRAPHS.

6.6 Le langage de description graphique

Le langage de description graphique est l'ensemble des concepts qui permettent de décrire comment doivent être représentés des relations et des types d'objets. Ce langage n'intègre pas uniquement les concepts graphiques de RECTANGLE, CERCLE etc... En effet, pour ces concepts il est possible de donner un certain nombre de paramètres qui décrivent par exemple le texte à inclure et la façon de représenter ce texte à l'intérieur du concept. Ces paramètres sont les suivants :

- le texte que l'on veut afficher dans le concept graphique :

exemple 6.6.1 : RECTANGLE (concat ["toto", texte])

qui aura pour effet d'afficher un rectangle dans lequel se trouvera le texte "toto" concaténé avec le texte contenu dans la variable texte.
L'opération concat est utilisée pour spécifier la concaténation, et les symboles "" pour indiquer qu'il s'agit de texte à représenter tel quel.

- la façon dont on veut que le texte soit affiché dans le concept graphique :

les modes de composition spatiale permettent de spécifier la façon dont le texte doit être agencé dans le concept graphique. Ils utilisent le concept de boîte qui symbolise la façon dont le texte est représenté, à savoir dans une boîte. Les modes de composition présentés à la section 6.5 sont utilisables, excepté I (intérieur). Les modes de composition suivants sont ajoutés EES (EetS), EOS (EouS) qui correspondent en fait respectivement aux concepts HetV et HouV présentés à la section 6.2. La signification de ces modes de composition est la suivante :

+ EOS (HOV): alignement horizontal si place suffisante sinon alignement vertical,
+ EES (HV): alignement horizontal avec passage à la ligne si la place n'est pas suffisante en largeur.

Ces notions sont surtout utiles lorsqu'il s'agit d'afficher plusieurs lignes de texte dans un concept graphique représentant une forme géométrique fermée.

exemple 6.6.2 : RECTANGLE (concat ["if", *COND] [EES.1]
concat ["then", *STAT1] [EES.1]
concat ["else", *STAT2])

Les représentations graphiques possibles sont présentées à la figure 6.6.1 ou 6.6.2. Dans la première figure, la place occupée en largeur par le concept graphique a été minimisée. A cet effet, deux passages à la ligne ont été effectués puisque le mode de composition EES a été spécifié. Par contre, dans la deuxième figure, la place occupée en largeur n'a pas été minimisée et donc aucun passage à la ligne n'a été effectué.

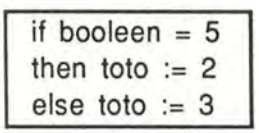


Figure 6.6.1 : représentation graphique possible de l'exemple 6.6.1

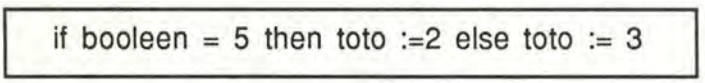


Figure 6.6.2: représentation graphique possible de l'exemple 6.6.1

Remarques

1. Il serait aussi intéressant de proposer à l'utilisateur de pouvoir combiner des concepts graphiques élémentaires pour former des concepts graphiques plus élaborés.
2. Une autre extension possible est d'envisager l'utilisation d'un éditeur graphique pour la création par l'utilisateur de formes géométriques. Ces formes devraient être stockées de manière géométrique et non pas sous forme de bitmaps puisqu'il faut avoir la possibilité de les agrandir ou de les diminuer suivant la place disponible lors de l'affichage.

6.7 Module de saisie

Ce module sera chargé d'obtenir

- le type d'objet pour lequel l'utilisateur veut une représentation graphique,
- la relation que représenteront les liens graphiques entre les différents objets représentés graphiquement.

Note : le type d'objet ne devra pas être introduit de manière explicite par l'utilisateur. Celui-ci pointera simplement sur un objet de ce type.

Spécification pré-post du module de saisie

saisie (TDOR, TYPE_OBJET, RELATION, NOEUD_ARBRE_ABSTRAIT)

- argument :
  TDOR : table de description des objets et des relations

- pré-condition :

- résultats :
  TYPE_OBJET : type de l'objet dont l'utilisateur veut une représentation graphique
  RELATION : relation choisie par l'utilisateur
  NOEUD_ARBRE_ABSTRAIT : noeud d'un arbre abstrait

- post-conditions :
  Le type d'objet TYPE_OBJET qui a été sélectionné par l'utilisateur et la relation RELATION à représenter graphiquement choisie par l'utilisateur sont renvoyés. Si le type d'objet TYPE_OBJET n'est pas présent dans la TDOR le message d'erreur < Aucune relation n'a été définie pour le type d'objet TYPE_OBJET > sera affiché à l'écran. De plus le noeud de départ dans l'arbre abstrait qui correspond au noeud que l'utilisateur a sélectionné NOEUD_ARBRE_ABSTRAIT sera renvoyé.

6.8 Module de décompilation graphique

Ce module devra effectuer les opérations suivantes :

- à partir de la saisie obtenue par le module de saisie, consulter la T.D.O.R. pour voir ce qu'il y a lieu de faire,
- guidé par les informations obtenues dans la T.D.O.R. aller chercher l'information dans l'arbre abstrait et construire la structure intermédiaire sans notion de positions.

Dès lors ce module fera appel à un module d'accès aux tables (en particulier pour un accès à la table T.D.O.R.) et appellera le module de recherche de l'information et de construction de la structure intermédiaire. Nous présentons d'abord le module d'accès aux tables et ensuite le module de construction de la structure intermédiaire.

6.8.1 Module d'accès aux tables

Ce module sera chargé d'effectuer les consultations dans les tables. Ce module va avoir deux fonctionnalités :

1) vérifier qu'un argument donné fait partie d'une colonne spécifiée dans une table (par exemple la T.D.O.R.)

Spécification pré-post

appartient (TABLE, COMPOSANT, NOM-COLONNE, NO-LIGNE)

- arguments :
  TABLE : table (par exemple T.D.O.R.)
  COMPOSANT : argument dont il faut vérifier s'il appartient à la table TABLE
  NOM-COLONNE : nom de la colonne de la table TABLE dans laquelle il faut vérifier l'appartenance de l'élément COMPOSANT

- pré-condition :
  NOM-COLONNE est un nom de colonne qui existe dans la table TABLE

- résultat :
  NO-LIGNE : numéro de la ligne de la table TABLE sur laquelle figure l'élément COMPOSANT dans la table TABLE

- post-conditions :
  (NO-LIGNE > 0 et COMPOSANT $\in$ NOM-COLONNE (TABLE)) ou (NO-LIGNE = 0 et COMPOSANT $\notin$ NOM-COLONNE (TABLE))

2) obtenir un élément d'une colonne de la table en précisant cet élément par le numéro de la ligne et le nom de la colonne de la table à laquelle il se trouve

Spécification pré-post

consult (TABLE, NO-LIGNE, NOM-COLONNE, COMPOSANT)

- arguments :
  TABLE : table
  NO-LIGNE : numéro de la ligne de la table à laquelle il faut consulter
  NOM-COLONNE : nom de la colonne de la table à laquelle il faut consulter

- pré-conditions :
  NO-LIGNE > 0
  NOM-COLONNE est un nom de colonne qui existe dans la table TABLE

- résultat :
  COMPOSANT : élément de la table

- post-conditions :
  COMPOSANT est l'élément de la table TABLE se trouvant à la ligne NO-LIGNE et à la colonne NOM-COLONNE

Ces deux modules vont être utilisés pour tous les accès aux tables (table de description des objets et des relations, table des concepts graphiques, table des correspondances entre structures logiques et physiques : voir section 6.9)

6.8.2 Module de recherche de l'information dans l'arbre abstrait et de construction de la structure intermédiaire

Ce module sera chargé d'aller chercher l'information nécessaire dans l'arbre abstrait et de construire la structure intermédiaire sans notion de positions.

Pour la recherche dans l'arbre abstrait, l'algorithme sera guidé par les informations obtenues dans la T.D.O.R. grâce au module d'accès. Il connaîtra aussi le noeud de départ duquel il doit partir dans l'arbre abstrait ainsi que le noeud racine de l'arbre abstrait. Il pourra alors appliquer la définiton (algorithme de parcours de l'arbre) de la relation tout en construisant la structure intermédiaire sans notion de positions, dûment remplie.

Spécification pré-post de ce module

recherche_construction (ARBRE_ABSTRAIT, NOEUD_DEPART, NOEUD_ RACINE, TDOR, STRUCT_INT)

- arguments :
  ARBRE_ABSTRAIT : arbre abstrait dans lequel il faut aller chercher l'information
  NOEUD_DEPART : noeud de départ dans l'arbre abstrait duquel il faut partir
  NOEUD_RACINE : noeud racine de l'arbre abstrait
  TDOR : table de description des objets et des relations

- pré-condition :

- résultat :
  STRUCT_INT : structure intermédiaire sans notion de positions

- post-conditions :
  STRUCT_INT contient toutes les informations nécessaires (cfr 6.10), excepté les positions pour l'affichage de la représentation graphique souhaitée par l'utilisateur et donnée par les colonnes D.F. et C de la T.D.O.R. (cfr section 6.5.2).

6.9 Table des correspondances entre structures logiques et physiques

Dans cette table, il faut indiquer les correspondances entre les structures logiques utilisées dans le langage de description d'objets et de relations inter-objets et les structures physiques (dans un langage de programmation) effectivement utilisées pour un arbre abstrait pour représenter ces structures logiques.
Par exemple considérons la structure logique table.

Supposons que les tables soient représentées en LISP au moyen de A-Listes. Les A-listes (listes d'associations) sont des tables qui possédent la structure suivante :
((clef1.val1) (clef2.val2) .. (clefN.valN)). Chaque élément d'une A-liste est un couple constitué d'une clef et d'une valeur. L'accès à une valeur est réalisée au moyen de sa clef.

Une opération logique possible sur la structure logique table est l'opération d'accès à un élément dans une table en donnant son indice. Cette opération logique sera implémentée au niveau physique par la fonction LE_LISP **ASSQ.** La fonction ASSQ < symb > < al > retourne l'élément de la A-liste < al > dont la clef est égale au symbole < symb >. L'algorithme de recherche dans l'arbre abstrait et de construction de la structure intermédiaire, extrait l'information dans un arbre abstrait en étant guidé par la colonne D.F. de la T.D.O.R. Chaque fois qu'il rencontre dans la colonne D.F. de la T.D.O.R. une opération sur une stsucture logique, il doit connaître l'opération "physique" à appliquer à cet arbre abstrait pour obtenir l'information requise. En fait, seules les opérations possibles sur les différentes structures logiques utilisées dans le texte de description doivent se retrouver dans cette table. Cela présente les avantages suivants :

- pouvoir "adapter" ces structures logiques aux structures physiques utilisées concrètement (dans un langage de programmation) pour représenter ces structures logiques,

- créer une librairie de correspondances entre structures logiques et structures physiques,

- permettre la construction de nouvelles correspondances logiques-physiques.

L'algorithme de recherche dans l'arbre abstrait devra consulter cette table de correspondance chaque fois qu'il rencontre une opération sur une structure logique.

Les opérations doivent être décrites comme suit dans la table des correspondances :

nom-opération	procédure-à-appeler
acct	ACCES_TABLE_CODE

Les conditions à respecter sont les suivantes :

- les "nom-opération" doivent être identifiantes,
- la "procédure-à-appeler" sera appelée avec les paramètres dans l'ordre dans lequel ils apparaissent dans le texte de description,
- la procédure doit renvoyer une liste de noeuds sous un certain format obligatoire connu de l'algorithme de recherche qui se servira de cette liste de noeuds. Cette liste pourra éventuellement être vide,
- lors de l'utilisation de l'opération dans le texte de description :
  - les paramètres de l'opération doivent être séparés par des virgules,
  - les paramètres doivent figurer entre des délimiteurs,

  Il y a deux possibilités :
  - délimiteur fixé qui doit toujours être utilisé quelle que soit l'opération,

    exemple : *délimiteur-début* = [et *délimiteur-fin* =]
  - laisser la possibilité de définir pour chaque opération le délimiteur à utiliser. Ce délimiteur doit alors être présent dans la table de correspondance, grâce aux deux paramètres *délimiteur-début, délimiteur-fin,*
- pour chaque nom-opération une procédure doit y être associée.

  Chaque procédure devra être construite suivant les règles énoncées ci-dessus. Au niveau implémentation, il pourrait y avoir un problème dans la mesure où chaque procédure doit renvoyer une liste de noeuds. Les noeuds pouvant être de type différent et de structure différente d'un arbre abstrait à l'autre, la procédure renverra une liste de pointeurs vers les noeuds. De plus pour certaines structures logiques, comme par exemple la structure logique (constructeur de types) PC définie en 6.13 il s'agira dans certains cas d'une liste de valeurs. Donc, ce sera toujours une liste de pointeurs vers l'information qui sera renvoyée.

exemple :

Supposons la définition des opérations sur la structure logique LIST-NOEUD qui représente une liste de noeuds, mais d'un même type.
Soit LIST-NOEUD1 une liste de noeuds de type BLOC-TYPE
et LIST-NOEUD2 une liste de noeuds de type BLOC-REL
Une opération utile sur cette structure logique est l'opération acc dont la spécification est la suivante :
LISTE-RES = acc (LISTE, I)
arguments :

I : rang dans la liste de l'élément que l'on veut obtenir
LISTE : liste de noeuds

résultat :

LISTE-RES : liste de noeuds

pré-condition :

post-condition :

(0 ≤ I ≤ taille (LISTE) et LISTE-RES = LISTE (i))

ou ((I = 0 ou I > taille (LISTE)) et LISTE-RES = ∅)

Il est clair que suivant l'application de l'opération acc à la liste de noeuds de type BLOC-TYPE ou à la liste de noeuds de type BLOC-REL, la liste résultat LISTE-RES contiendra un noeud de type différent.

6.10 La structure intermédiaire graphique (S.I.G.)

6.10.1 Introduction

La S.I.G. permet de résoudre le conflit de structures entre la représentation interne (arbre abstrait) et externe (écran) de la relation.
L'utilisateur doit pouvoir faire certaines opérations sur la représentation externe graphique. Ce qui justifie la présence de la S.I.G. qui de par sa **structure physique** doit permettre la réalisation de ces opérations. Nous présentons, en premier lieu, le choix opéré concernant la structure physique de cette S.I.G., ainsi que l'information qu'elle doit contenir. Ensuite nous présentons les différents problèmes qu'engendrent les opérations qu'il doit être possible d'effectuer sur la S.I.G., à savoir, le problème du **positionnement**, le problème de la **désignation** et enfin celui de la **modification**. Nous terminons cette section par une évaluation critique du choix proposé pour cette S.I.G..

6.10.2 Rappels sur les graphes

Toutes ces définitions proviennent des notes de cours de [Fichefet 85], et du mémoire de [Lefèvre 87] sur l'interprétation des mesures de performance d'un système.

6.10.2.1 Graphes orientés

Un **graphe orienté** G est un couple G = (X,U) constitué par

- un ensemble $X = \{x_1, x_2, ..., x_n\}$ fini ou dénombrable dont les éléments sont les **sommets** du graphe.
- une famille $U = \{u_1, u_2, ..., u_n\}$ de couples de sommets appelés les **arcs** du graphe.

Extrémité initiale et terminale d'un arc

Soit G = (X,U) un graphe avec $X = \{x_1, x_2, ...,x_n\}$. Si $u = (x_i, x_j)$ appartient à U, on dit que x_i est l'**extrémité initiale** et que x_j est l'**extrémité terminale** de l'arc.

Arbre

Soit A = (X,U), A est un **arbre** si aucun sommet de A n'est extrémité terminale de plus d'un arc.

Chemin, longueur d'un chemin

Un **chemin** est une suite $\mu = (u_1, u_2, ..., u_i, u_{i+1}, ..., u_q)$ d'arcs tels que l'extrémité terminale d'un arc u_i coïncide avec l'extrémité initiale de l'arc suivant u_{i+1}.

Le nombre q d'arcs qui composent la suite μ est la longueur du chemin μ.

Précédent, suivant

Le sommet x_i est un **précédent** du sommet x_j si x_i est l'extrémité initiale d'un arc dont x_j est l'extrémité terminale, auquel cas, le sommet x_j est **suivant** du sommet x_i.

Ascendant, descendant, racine

Le sommet x_i est un **ascendant** du sommet x_j s'il existe un chemin ayant x_i et x_j comme extrémités initiale et terminale.
Le sommet x_j est un **descendant** du sommet x_i lorsque x_i est ascendant de x_j.
Le sommet x_j est **accessible** du sommet x_i, si x_j est descendant de x_i. Un sommet qui est descendant de tout sommet du graphe est **racine** du graphe.

P-graphe

Lorsqu'au plus p arcs vont d'un sommet quelconque à un autre sommet quelconque, le graphe est appelé **p-graphe**.

Circuit

Un **circuit** est un chemin dont les extrémités coïncident.

Fermeture transitive

Un **1-graphe** peut également être défini par un couple (X,A) où X est l'ensemble des sommets et où A est une application multivoque de X dans l'ensemble P(X) de ses parties, qui, à tout sommet $x_i \in X$, fait correspondre le sous-ensemble Ax_i est inclus dans l'ensemble X des sommets reliés à x_i par un arc en provenance de xi.
La **fermeture transitive** de A est l'ensemble de tous les descendants du sommet x_i.

Matrice d'accessiblité

Soit M la **matrice d'accessibilté** du graphe, un élément m_{ij} indique le nombre de chemins joignant le sommet x_i au sommet x_j.

Composantes fortement connexes (C.F.C.)

Une **C.F.C.** est un ensemble de sommets X tels que pour tout sommet x_i et x_j appartenant à X, il existe un chemin allant de x_i à x_j et il existe un chemin allant de x_j à x_i.
Par conséquent, deux sommets x_i et x_j font partie d'une C.F.C. s'il existe un circuit passant par x_i et x_j ou si $x_i = x_j$.

6.10.2.2 Graphes hiérarchisés

Hiérarchie, niveaux

On appelle **hiérarchie** H sur un graphe G = (X,U) toute application de l'ensemble des sommets X = {x_i, i = 1, ..., m} dans une partie I = {1, 2, ..., k} de l'ensemble des naturels, associant à tout sommet x_i appartenant à X une valeur n_i appartenant à I.
On dit que le sommet x_i est au **niveau** n_i dans la hiérarchie H, et k est le nombre de niveaux de la hiérarchie.

Hiérarchie des rangs

Il existe une hiérarchie particulière dans le cas des graphes **sans circuit** appelée **hiérarchie des rangs** qui, à chaque sommet, fait correspondre le rang de ce dernier augmenté de un.
Un sommet s est dit de **rang r**, si et seulement si, r est le nombre maximal d'arcs de tout chemin d'extrémité terminal s (un sommet sans précédent est donc de rang 0).

Portée d'un arc

La **portée d'un arc** (x_i, x_j) est la différence n_j - n_i des niveaux de ses extrémités.

Hiérarchie serrée

On appelle **hiérarchie serrée**, une hiérarchie compatible avec l'ordre des rangs (c'est-à-dire que si on prend deux sommets x_i et x_j de niveaux respectifs n_i et n_j tels que $n_i > n_j$ le rang de x_i est > au rang de x_j) et qui conduit à une représentation telle que la portée de tous les arcs est minimale.

Hiérarchie propre

Si tous les arcs dans une hiérarchie ont une portée égale à un, on dit que la **hiérarchie est propre**.

Longueur, largeur, mesure d'une hiérarchie

Le nombre de niveaux d'une hiérarchie est appelé la **longeur** d'une hiérarchie et le nombre maximum de sommets d'un niveau, la **largeur** d'une hiérarchie. Le nombre d'arcs est appelé la **mesure** d'une hiérarchie.

Chemin maximum

Un **chemin maximum** est un chemin contenant autant de sommets qu'il y a de niveaux dans la hiérarchie.

Planarité d'un graphe

Un graphe est **planaire** s'il est susceptible d'être dessiné dans un plan sans que ses arêtes ne se coupent.

6.10.3 La structure physique de la S.I.G.

6.10.3.1 Problématique

Avant d'exposer les différentes structures physiques possibles, il est nécessaire de présenter les exigences qu'elles doivent respecter.
Un texte a une structure bien spécifique qui fait qu'en général, on le lit de haut en bas et de gauche à droite. Disposer le texte d'une autre manière, le rend presque illisible. Il suffit pour s'en convaincre, d'afficher le texte en partant de haut vers le bas mais de droite à gauche.
Notre S.I.G., comme son nom l'indique, "contient" une représentation graphique. Celle-ci a comme particularité par rapport à la représentation textuelle, de ne pas avoir une disposition prédéfinie. Plus précisément, elle a une structure prédéfinie mais dans notre cas c'est l'utilisateur qui la définit grâce au L.D.O.R., ce qui fait qu'elle peut varier d'un utilisateur à l'autre. Cependant, l'utilisateur doit pouvoir la définir comme bon lui semble. Il peut vouloir disposer un concept graphique à gauche de l'écran, un autre en bas etc., sans pour autant adopter un parcours de lecture haut - bas, gauche - droite.
Comme la S.I.G. reflète ce qui est à l'écran, celle-ci doit permettre de représenter une telle disposition. La structure intermédiaire textuelle comme celle du système SACSO n'est pas suffisante pour une représentation graphique.
De plus, les différents objets graphiques peuvent être reliés entre eux par d'autres concepts graphiques. Ils peuvent également former une composition d'objets. Enfin, nous devons considérer qu'en toute généralité, la relation modélisée dans l'arbre abstrait peut donner une représentation externe graphique sous forme de graphe.

6.10.3.2 La S.I.G. basée sur le concept de boîte

Nous présentons une structure intermédiaire dont la structure est calquée sur le modèle de boîte [Coutaz 85] auquel nous avons ajouté quelques concepts pour l'adapter à une représentation graphique. Nous ne revenons pas sur ce concept de boîte qui a déjà été expliqué dans la section 6.2. Nous présentons plutôt les concepts introduits. Ensuite, nous expliquons pourquoi elle ne convient pas.
Dans cette structure, les objets sont des boîtes. La S.I.G. a la forme d'un arbre de boîtes. A l'aide de l'attribut de composition des boîtes (cfr. 6.2), nous pouvons disposer les objets graphiques. Par exemple, un objet graphique peut se trouver à la verticale ou à l'horizontale d'un autre objet graphique. Les boîtes H, V permettent cette composition de boîtes.
Pour exprimer les liaisons entre les boîtes, nous ajoutons un attribut supplémentaire exprimant le type de liaison : DROITE, DROITE-ANGLE, etc. Cette liaison ne concerne que les boîtes qui sont descendantes des boîtes H, V. Pour permettre une disposition plus générale des boîtes, nous ajoutons d'autres attributs de composition qui sont les attributs EST, OUEST, NORD, SUD ainsi que leurs combinaisons NORD-EST, NORD-OUEST, etc. (cfr. 6.5.1). Nous illustrons ce fait par la figure 6.10.1 qui nous donne la S.I.G. et sa représentation externe graphique correspondante.

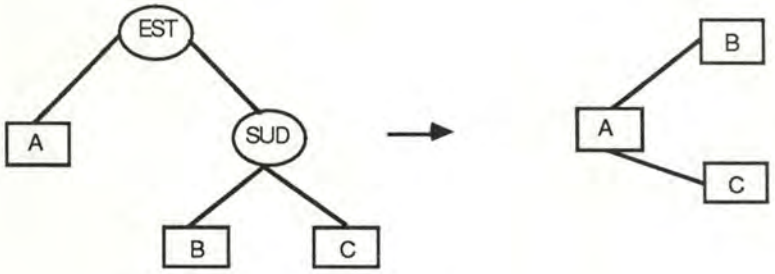


Figure 6.10.1 : S.I.G. et la représentation externe

La boîte dont l'attribut de composition est EST, exprime que le sous arbre de droite est à l'EST du sous arbre de gauche. La boîte C se trouve au SUD de la boîte B et à l'est de la boîte A.
Ce petit exemple illustre le fait que les boîtes de la branche de droite se positionnent par rapport aux boîtes de la branche de gauche en l'occurence la boîte A. Cette boîte de référence, ou "boîte de départ" est généralement la boîte "mère" de la liaison avec les boîtes filles. Les boîtes B et C représentent donc les boîtes filles qui sont positionnées à l'EST de la boîte A. La figure 6.10.1 illustre également le fait que sur un même niveau de l'arborescence, il ne peut y avoir au maximum que deux boîtes. Aussi, une telle structure ne nous permet pas d'exprimer par exemple qu'une boîte fille se trouve à l'EST et qu'une autre se trouve au NORD de la boîte mère.
En effet, quelle serait la S.I.G. permettant une telle représentation (figure 6.10.2)?

Figure 6.10.2 : quelle S.I.G.?

Par contre, si nous considérons la boîte B ou C comme la boîte mère, nous pouvons donner les S.I.G. (figure 6.10.3) représentant le dessin précédent.

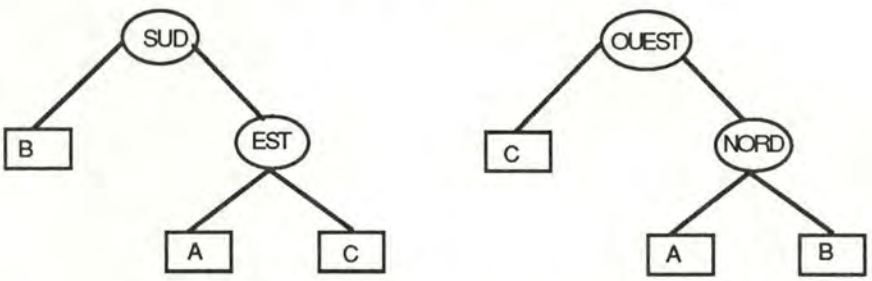


Figure 6.10.3 : S.I.G. avec la boîte B ou C comme boîte mère

Si nous considérons une disposition étoilée des boîtes, il est possible, en ayant sous les yeux la représentation, de trouver une structure qui la décrit. Il suffit de trouver la boîte de départ. Cependant, quand il s'agit d'automatiser la construction de cette structure interne à partir d'une description par un utilisateur, nous n'avons pas le dessin sous les yeux. Ce qui fait qu'il n'est

pas possible de choisir la boîte de départ en fonction du dessin. Nous avons donc des cas de figures qui sont impossibles à représenter. Il suffit d'avoir deux boîtes filles non reliées entre elles, qui soient positionnées par rapport à la boîte mère par des boîtes dont les attributs de composition ne sont pas combinés entre eux. Par exemple, une boîte fille située au NORD, et une autre située au SUD forment un cas de figure non représentable. Par contre, une boîte fille située au SUD et une autre située au SUD-EST forment un cas de figure représentable puisque les attributs de composition sont combinés.
Nous voyons donc qu'une structure de boîte étendue (EST, OUEST, etc.) ne convient pas pour une représentation graphique.
Le problème semble dû au fait que nous ayons deux notions différentes exprimées dans une même boîte. La boîte de composition (EST, OUEST, etc.) nous donne les boîtes à positionner, ce sont les boîtes "suivantes" dans la S.I.G.. Mais elle exprime également leur position.
C'est pourquoi, dans la solution proposée, nous séparons et mettons sur un même niveau, les boîtes à positionner et les boîtes qui les positionnent.

6.10.3.3 Solution proposée

Notre S.I.G. se présente sous la forme d'une **arborescence de boîtes typées**. Cette arborescence exprime une **hiérarchie de composition spatiale**. Une boîte qui, dans la hiérarchie est "suivante" d'une autre, est une composante de celle-ci.
Les boîtes de type **inter-boîte** permettent le placement de ces boîtes composantes l'une par rapport à l'autre. Aussi nous avons des inter-boîtes dont l'**attribut de composition** est SUD, NORD, EST, OUEST, SUD-EST, etc. (cfr. 6.5.1). Les inter-boîtes permettent également de faire abstraction du concept graphique qui relie deux boîtes composantes entre elles. Les inter-boîtes sont des feuilles de l'arborescence.

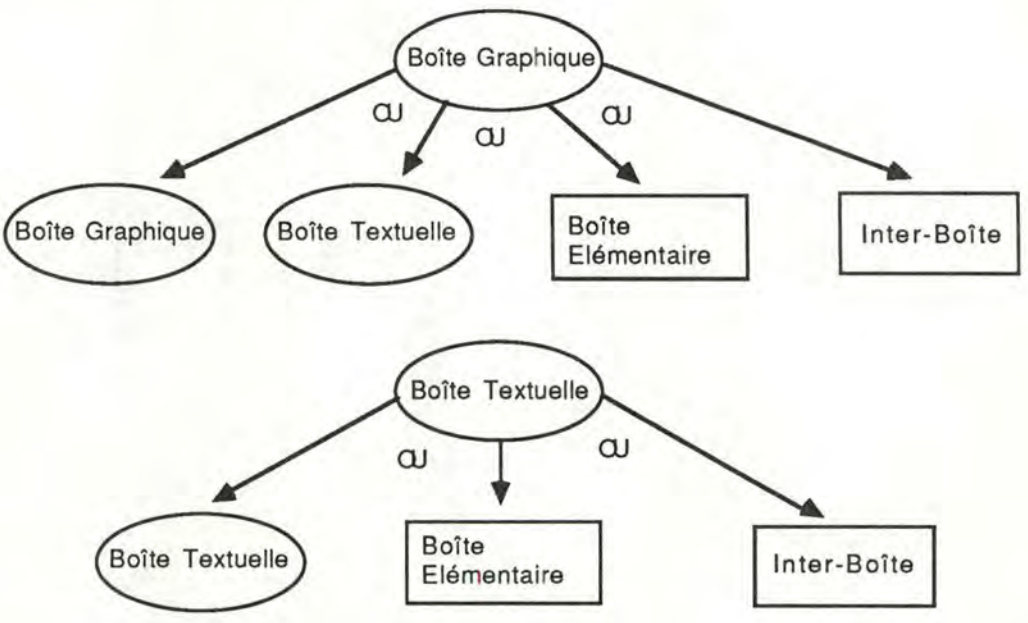


Figure 6.10.4 : structure de la S.I.G. proposée

La boîte composition graphique, ou **boîte graphique** fait abstraction du concept graphique représenté par cette boîte. Une boîte graphique peut être composée de boîtes graphiques,

d'inter-boîtes, et/ou de boîtes composition textuelles (ou boîtes textuelles). La **boîte textuelle** peut être composée de boîtes textuelles, d'inter-boîtes et/ou de **boîtes élémentaires**.
Les boîtes élémentaires sont également des feuilles de l'arbre, elles représentent les éléments ou unités de texte composées entre elles pour former la boîte textuelle.
Une inter-boîte peut exprimer le placement de boîtes qui ne sont pas boîtes composantes d'une même boîte composée. Ceci afin de pouvoir représenter un graphe à l'écran.
Nous pouvons présenter la structure physique de l'arborescence à l'aide de la figure 6.10.4. Dans celle-ci, tout noeud de la S.I.G. représente une boîte typée. Les rectangles sont les feuilles, tandis que les éllipses sont les noeuds non-terminaux. Chaque flèche ou arc de la S.I.G. représente une composition de l'extrémité initiale de l'arc par l'extrémité terminale. Entre les arcs, nous avons une relation "ou" non exclusive.

A titre d'illustration de la S.I.G., nous donnons la figure 6.10.5 suivante :

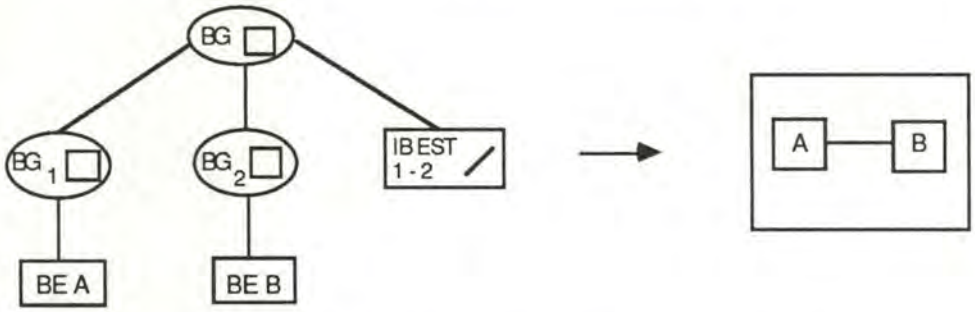


Figure 6.10.5 : S.I.G. et la représentation externe graphique

Dans cet exemple, la boîte racine de la S.I.G., est une boîte graphique dont le concept est rectangle. Elle est composée de boîtes graphiques reliées par une inter-boîte EST dont le concept graphique est droite. Pour l'exemple, les identifiants des boîtes graphiques sont représentés par un numéro. Cependant, dans la S.IG. toute boîte dispose d'un identifiant et de plus, celui-ci a une structure plus complexe.
Nous ne donnons ici, que la structure physique générale de la S.I.G.. Dans la section 6.10.4, le contenu de chaque noeud est développé.
Les boîtes composantes d'une même boîte composée sont situées sur un même niveau, ce qui exprime le **caractère plat** de la S.I.G.. Cela nous permet d'éviter les problèmes de "boîte de départ" rencontrés précédemment. Cette caractéristique de la S.I.G. apporte des avantages et des inconvénients. Nous les énoncons dans la section 6.10.8.

Pour faciliter la description donnée par l'utilisateur, l'inter-boîte est considérée comme une feuille de l'arborescence. Nous pourrions aussi la considérer comme une boîte composition, ce qui permettrait d'une manière élégante, de la composer avec des boîtes graphiques pour former de nouvelles inter-boîtes. Les possibilités de liaison graphique en seraient considérablement augmentées. En même temps, nous diminuerions le nombre de concepts graphiques différents que doit fournir le système, car certains concepts graphiques pourraient être obtenus à partir de combinaisons de concepts plus élémentaires.
En reprenant la structure physique de la S.I.G., l'inter-boîte serait alors représentée par une boîte composition (figure 6.10.6):

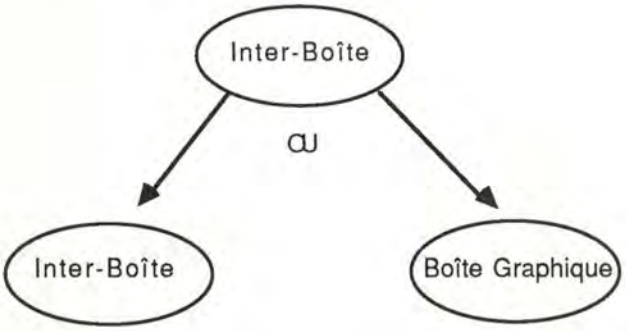


Figure 6.10.6 : inter-boîte composée

Voici à titre d'illustration, quelques figures d'inter-boîtes qu'il serait possible de combiner à partir des concepts graphiques droite, cercle, rectangle.

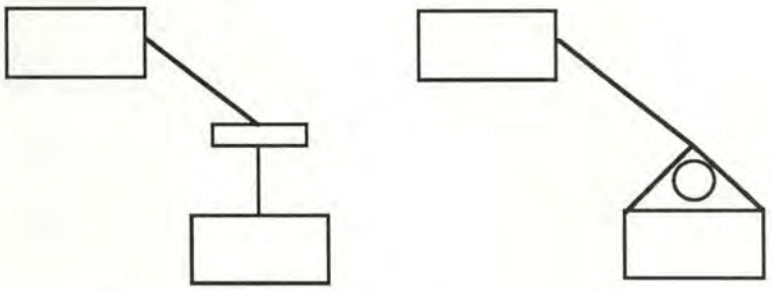


Figure 6.10.7 : illustrations d'inter-boîtes

Une deuxième remarque peut être apportée concernant les boîtes textuelles. Nous aurions pu également utiliser la structure d'arbre proposée dans [Coutaz 85] et qui convient parfaitement bien pour une représentation textuelle. La boîte textuelle aurait un attribut de composition H, V, HouV, HetV, et nous n'aurions plus d'inter-boîtes composantes de cette boîte textuelle. Une telle structure se justifie lorsque le problème de l'indentation du texte affiché est important et qu'il faut une structure qui puisse l'optimiser. Ce qui n'est pas notre cas, puisque le texte affiché a pour seul but de clarifier les concepts graphiques. Ainsi, dans un souci de généralité, nous avons adapté pour ces boîtes textuelles, le même modèle Boîte-Inter-boîte que celui appliqué aux boîtes graphiques. Cependant, nous ajoutons aux inter-boîtes, les attributs de composition ESTouSUD, ESTetSUD (cfr. 6.6) s'appliquant uniquement aux boîtes textuelles. Ils donnent la possibilité, d'afficher du texte en passant à la ligne lorsqu'il n'y a pas assez de place sur la largeur.
Il est également important de noter que la structure physique de la S.I.G. n'est pas l'image fidèle de ce qui est représenté à l'écran. C'est une structure arbre tandis que la représentation externe graphique peut être un graphe non planaire. Les arcs de la S.I.G. ne correspondent pas aux arcs du graphe représenté. Ces derniers sont en effet représentés par les inter-boîtes.

6.10.4 Information contenue dans un noeud de la S.I.G.

Nous présentons sous la forme de champs, toute l'information que doit contenir un noeud de la S.I.G.. Suivant le type des boîtes, certains champs du noeud ne sont pas exploités. D'autres font référence à des notions qui sont expliquées ultérieurement, mais nous en donnons tout de même une brève description.

6.10.4.1 Le champ TYPE-BOITE

Nous avons quatre types de boîtes qui sont la **boîte graphique**, la **boîte textuelle**, l'**inter-boîte**, et la **boîte élémentaire**.
Cette information est primordiale car elle permet une interprétation cohérente de la S.I.G. compte tenu de sa syntaxe présentée auparavant.
Elle permet également d'exploiter correctement les différents champs de cette S.I.G.. Par exemple, dans le cas d'une boîte élémentaire, le champ ATTRIBUT-COMPOSITION ne doit pas être considéré.

6.10.4.2 Le champ IDENTIFIANT-BOITE

Chaque boîte dispose d'un **identifiant**. Cet idendifiant est formé de trois parties.
1) Le **numéro-niveau** : donne le niveau de la boîte dans la hiérarchie de composition spatiale. La boîte de niveau zéro est la racine, ensuite au fur et à mesure que nous descendons dans la hiérarchie, le niveau augmente.

2) Le **numéro-boîte** : donne un numéro qui identifie la boîte parmi toutes les boîtes composantes d'une même boîte composée. Ce numéro correspond à la numérotation des suivants de la boîte.
Si le numéro-boîte exprimait le nombre de boîtes qu'il y a sur un niveau, les deux champs numéro-niveau et numéro-boîte suffiraient pour être identifiant de la boîte. Cependant, la modification de la S.I.G. pourrait entraîner une mise à jour des identifiants pour toutes les boîtes situées sur le même niveau que la boîte modifiée. De plus, au niveau de la construction de la S.I.G., avant d'attribuer un numéro-boîte, il faut connaître le nombre de boîtes qui existent déjà sur le niveau.
C'est pourquoi, nous prenons un numéro-boîte exprimant seulement le nombre de boîtes composantes d'une boîte composée. Ainsi, sur un niveau, deux boîtes différentes peuvent avoir le même numéro-niveau car elles ne sont pas composantes d'une même boîte composée. Nous devons donc introduire une troisième partie pour l'identifier.

3) Le **numéro-boîte-rattachée** : donne le numéro-boîte de la boîte composée à laquelle la boîte est rattachée.

A la figure 6.10.8, les lignes horizontales représentent les niveaux. A chaque boîte représentée par un cercle, nous associons son IDENTIFIANT-BOITE, dont l'ordre des parties est numéro-niveau, numéro-boîte, numéro-boîte-rattachée.

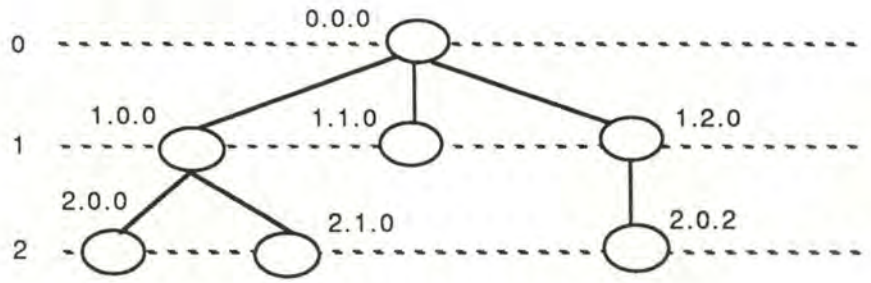


Figure 6.10.8 : identifiant d'une boîte

6.10.4.3 Le champ IDENTIFIANT-BOITE-MERE-UNITE

Le champ fait référence à un **concept d'unité** développé dans la section ultérieure sur les contraintes esthétiques. Ce champ a la structure d'une liste d'IDENTIFIANT-BOITE.

6.10.4.4 Le champ BOITE-REPRESENTEE

Il indique si la boîte est représentée ou pas à l'écran (expression du caractère **visible** de la boîte). Il permet de déterminer dans les boîtes de la S.I.G., celles qui n'ont pas encore été affichées à cause du manque de place à l'écran. Il est utilisé par l'algorithme d'annotation et par l'affichage.

6.10.4.5 Le champ BOITE-ABSTRACTION

Il indique si la boîte est une **boîte-abstraction.** Le champ est exploité par l'algorithme de désignation. Ce champ est nécessaire, car au niveau de la S.I.G., rien n'identifie une boîte-abstraction excepté ce champ. Il s'exprime sous forme de deux parties.

1) L'**indicateur** : indique si la boîte est une boîte abstraction ou pas.

2) Le **pointeur-boîte-abstraction** : pointe vers la boîte abstraction.

Ces deux champs seront expliqués par la suite (cfr. 6.11.3).

6.10.4.6 Les champs POINTEUR-BOITE-PRECEDENTE (SUIVANTE)

Ce sont des pointeurs qui permettent un parcours de l'arborescence dans les deux sens. **Remonter** par le pointeur-boîte-précédente, et **descendre** par la liste pointeur-boîte-suivante.

6.10.4.7 Le champ POINTEUR-ARBRE-ABSTRAIT

Ce champ permet la **correspondance** entre la S.I.G. et l'arbre abstrait. Il fait le **lien** entre la représentation graphique et la représentation textuelle par l'intermédiaire de l'arbre abstrait. Il définit en quelque sorte, le caractère "**sélectionnable**" d'une boîte de la S.I.G..

6.10.4.8 Le champ POINTEUR-FENETRE

Ce champ est un pointeur vers la **fenêtre** où est représentée la boîte. Une boîte d'une même S.I.G. ne se trouve que dans une et une seule fenêtre. Par contre la S.I.G. peut être éclatée dans plusieurs fenêtres.

6.10.4.9 Le champ COORDONNEES-BOITE

Ce champ exprime la **position écran** de la boîte. Nous l'expliquons de manière détaillée par la suite (cfr. 6.10.5).
Ce champ est composé de deux parties, l'une est consacrée à l'expression de coordonnées absolues et l'autre de coordonnées relatives. Ces parties sont exploitées par les algorithmes d'annotation et de désignation.

1) Les **coordonnées-absolues** : représentées par les coordonnées de deux points (coin supérieur gauche et coin inférieur droit).

2) Les **coordonnées-relatives** : représentées par deux fractions (abscisse et ordonnée relatives), et pour chacune, les champs numérateur, et dénominateur.

6.10.4.10 Le champ CONCEPT-GRAPHIQUE

Ce champ n'est initialisé que pour les boîtes graphiques ou les inter-boîtes. Il indique quel est le concept graphique qui y est représenté. Il est utilisé par les algorithmes d'annotation, de désignation et par l'affichage. Ce champ peut prendre les valeurs CERCLE, RECTANGLE, DROITE, etc., c'est-à-dire toutes les valeurs de la colonne nom-concept de la table des concepts graphiques (T.D.C.G.).
Pour les inter-boîtes, il peut prendre quelques valeurs particulières. Soit il est VIDE, ce qui signifie qu'entre les boîtes reliées, il y a un espace mais pas de concept graphique représenté. Soit il est COLLE, ce qui signifie qu'il n'y a pas d'espace entre les boîtes.

6.10.4.11 Les champs IDENTIFIANT-BOITE-EXTREMITE-INITIALE (TERMINALE)

L'inter-boite représente une liaison entre deux boîtes. Ces champs identifient les boîtes reliées. Ils ont chacun la structure d'un IDENTIFIANT-BOITE. La liaison peut être considérée comme un arc orienté qui va d'une boîte extrémité initiale à une boîte extrémité terminale.

6.10.4.12 Le champ ATTRIBUT-COMPOSITION

Il donne l'**orientation** de la composition. Cette orientation, initialisée dans une inter-boîte, exprime comment se place la boîte extrémité terminale par rapport à la boîte extrémité initiale de l'inter-boîte. Cette orientation peut prendre alors les valeurs des quatre points cardinaux, seuls ou combinés entre eux, ainsi que les valeurs EOS, EES spécifiques aux boîtes textuelles.

6.10.4.13 Le champ CONTENU-TEXTUEL

Ce champ est initialisé pour les boîtes élémentaires. Il donne l'**unité textuelle** qui doit être affichée.

6.10.4.14 Le champ RELATION-REPRESENTEE

Ce champ est défini pour **la modification incrémentale** de la S.I.G. Etant donné un noeud de l'arbre abstrait, celui-ci peut appartenir simultanément à différentes relations, modélisées dans l'arbre abstrait et représentées à l'écran. Lorsqu'une modification est opérée sur un noeud de l'arbre abstrait, nous devons répercuter la modification sur les différentes S.I.G. pour lesquelles ce noeud est représenté. Cependant, pour chacune de ces S.I.G., la modification est faite en fonction de la relation qui a permis leur construction. Ce champ permet d'éviter le recalcul complet de la S.I.G. correspondant à la relation.

En définitive, nous pouvons représenter un noeud de la S.I. comme une structure à seize champs :

- TYPE-BOITE
- IDENTIFIANT-BOITE
- IDENTIFIANT-BOITE-MERE-UNITE
- BOITE-REPRESENTEE
- BOITE-ABSTRACTION
- POINTEUR-BOITE-PRECEDENTE
- POINTEUR-BOITE-SUIVANTE
- POINTEUR-ARBRE-ABSTRAIT
- POINTEUR-FENETRE
- COORDONNEES-BOITE
- CONCEPT-GRAPHIQUE
- IDENTIFIANT-BOITE-EXTREMITE-INITIALE
- IDENTIFIANT-BOITE-EXTREMITE -TERMINALE
- ATTRIBUT-COMPOSITION
- CONTENU-TEXTUEL
- RELATION-REPRESENTEE

Ces champs représentent toute l'**information** nécessaire au niveau de la S.I.G..

Dans une optique plus générale, nous pouvons également proposer en tant qu'extensions du système, d'autres champs qui ont rapport aux caractères plus visuel et plus dynamique de la S.I.G..

6.10.4.15 Le champ MODE_PRESENTATION

Ce champ intègre des notions comme la **couleur**, l'**éclairage**, et le **style** (video inverse, etc.).

6.10.4.16 Le champ FORMATAGE

Ce champ intègre des notions comme les **contraintes d'espacement**, ou les **contraintes sur les dimensions** des boîtes.

6.10.4.17 Le champ ACTIONS

Ce champ représente l'**action** à effectuer lorsque la boîte reçoit un ordre d'activation correspondant par exemple à sa désignation par la souris. Par défaut, l'action effectuée est soit une décompilation textuelle, ou soit un affichage de la S.I.G. non encore représentée. D'autres actions comme celles offertes par un éditeur graphique, peuvent être intégrées dans ce champ. Par exemple, les actions déplacer, rendre invisible, ou faire abstraction de. Par analogie avec CEYX, nous pourrions associer des clés à ces différentes actions.

6.10.4.18 Le champ SELECTIONNABLE (MODIFIABLE)

Pour pouvoir effectuer une opération sur une boîte, il faut avant tout que celle-ci soit **sélectionnable.** Ce champ est plus générale que le champ POINTEUR-ARBRE-ABSTRAIT. En effet, une boîte n'ayant pas de lien avec un arbre abstrait peut tout de même

être sélectionnée pour effectuer une opération qui ne nécessite pas ce lien sémantique. Par exemple l'opération déplacer.
Pour pouvoir effectuer une opération modifiant la boîte comme par exemple l'opération agrandir, il faut que la boîte soit **modifiable**.

6.10.4.19 Le champ COMMENTAIRE

Ce champ un peu particulier, donne la possibilité d'associer du **commentaire** à chaque boîte. Ce champ fonctionne comme une "boîte aux lettres" mise à la disposition de l'utilisateur mais dont la gestion lui est totalement laissée. Le système se contente d'offrir les deux opérations élémentaires que sont la lecture et l'écriture dans la boîte aux lettres.

En annexe 5, nous donnons quelques exemples de S.I.G. ainsi que leur représentation externe graphique.

6.10.5 Coordonnées d'une boîte

L'annotation consiste à poser des coordonnées sur les boîtes de la S.I.G.. Nous allons définir dans cette sous-section ce que nous entendons par **coordonnées d'une boîte** lorsque nous nous situons dans la S.I.G..
Nous montrons en quoi l'annotation est **indépendante** des concepts graphiques utilisés dans la S.I.G. et définis dans **la table des concepts graphiques**.

Quelque soit le contenu de la boîte, texte ou graphique, nous supposons un **cadre imaginaire** qui l'englobe. Les dimensions de ce cadre sont déterminées par les dimensions du contenu ou contrainte par l'application (taille minimale, maximale de ce contenu). La position d'une boîte à l'écran est exprimée par les coordonnées écran (abscisse, ordonnée) des points **supérieur gauche** et **inférieur droit** du cadre (figure 6.10.9).

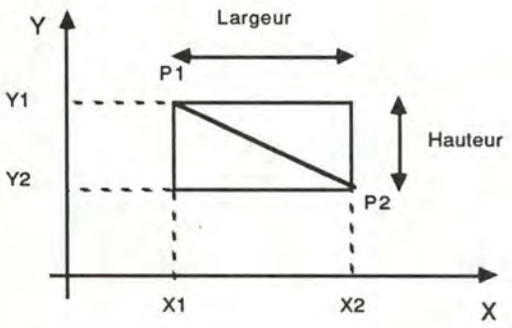


Figure 6.10.9 : coordonnées d'une boîte

La position écran de ce cadre dépend des critères énoncés dans la section suivante et des dimensions du contenu que ce cadre doit entourer.
Nous allons par type de boîte, déterminer le calcul des coordonnées de son cadre en fonction des dimensions du contenu de la boîte.

6.10.5.1 La boîte textuelle

TEXTE

Figure 6.10.10: la boîte textuelle

Les dimensions de cette boîte sont déterminées par les dimensions hauteur, et largeur de la chaîne de caractères "TEXTE". Connaissant la police de caractères utilisée et connaissant le texte qui est écrit, les dimensions de la boîte peuvent être calculées facilement. Etant donné un point par rapport auquel la boîte doit être positionnée et étant données les dimensions hauteur et largeur obtenues précédemment, nous pouvons calculer les coordonnées des points supérieur gauche et inférieur droit du cadre (points P1 et P2) :

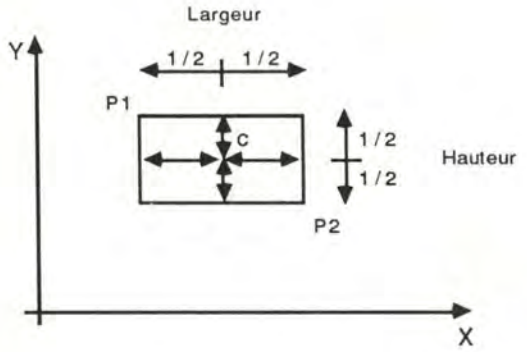


Figure 6.10.11 : dimensions de la boîte

Dans l'exemple, nous choisissons de positionner la boîte par rapport à un point centre C :

P1.abscisse = C.abscisse - largeur/2
P1.ordonnée = C.ordonnée + hauteur/2

P2.abscisse = C.abscisse + largeur/2
P2.ordonnée = C.ordonnée - hauteur/2

6.10.5.2 La boîte graphique

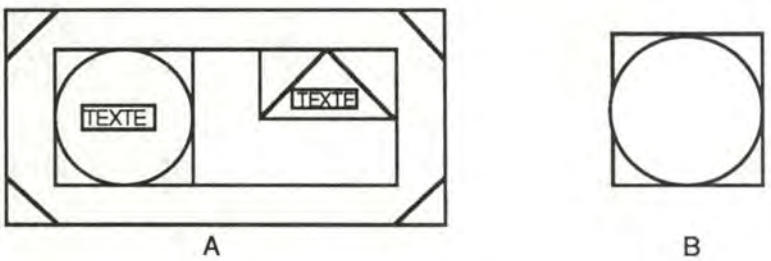


Figure 6.10.12 : la boîte graphique

Pour déterminer les dimensions de la boîte graphique, nous devons tenir compte des dimensions de la boîte composante et du concept graphique qui l'entoure, si concept graphique il y a.
Nous disposons d'une fonction appelée **fonction dimension** qui reçoit en paramètres la largeur et la hauteur de la boîte contenue et qui renvoit en résultats la largeur et la hauteur de la boîte, nécessaire pour contenir le concept graphique et ce qu'il entoure. Elle calcule en fonction du concept graphique et du contenu de la boîte, les dimensions du cadre. Pour obtenir les coordonnées des points P1 et P2 de la boîte, il suffit de positionner le cadre par rapport aux coordonnées d'un point donné.

Description de la fonction dimension

Cette fonction est spécifique au concept graphique qui entoure le contenu de la boîte. Comme il y a plusieurs concepts graphiques différents, il est nécessaire d'avoir une correspondance entre le concept graphique et sa fonction dimension. C'est pourquoi, nous avons défini cette correspondance dans **la table des concepts graphiques** (cfr 6.10.7).

Introduire un nouveau concept graphique se limite au niveau de cette table, à y ajouter sa description ainsi que sa fonction dimension correspondante.
Nous allons, pour quelques concepts graphiques considérés dans la table, décrire la fonction dimension qu'il convient d'utiliser.

Le cercle

Figure 6.10.13 : le cercle

Quels doivent être le centre et le rayon du cercle, pour que celui-ci puisse entourer le contenu représenté par le rectangle intérieur non hachuré?
Ce contenu peut également être une boîte graphique.
Il suffit de considérer le centre du contenu et sa diagonale.
Ce qui nous donne :

centre_cercle = centre_contenu
rayon_cercle = diagonale_contenu/2
et
largeur_boîte_graphique = rayon_cercle * 2
hauteur_boîte_graphique = rayon_cercle * 2

Le rectangle

Pour un rectangle, le problème est trivial. Il suffit de considérer :

largeur-boîte-graphique = largeur-contenu
hauteur-boîte-graphique = hauteur-contenu

Le triangle

Supposons un triangle isocèle.

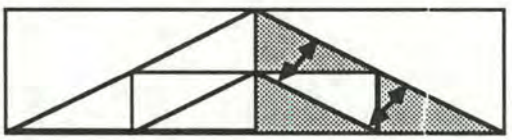


Figure 6.10.14 : le triangle

Considérons la boîte contenue représentée à la figure 6.10.14 par un rectangle. Dans cette boîte, nous pouvons y dessiner un triangle isocèle dont la hauteur est égale à la hauteur du rectangle et dont la base est égale à la largeur de ce rectangle. Le triangle isocèle entourant le rectangle, peut être obtenu par parallélisme (homotétie) avec le triangle inclus dans le rectangle. Ce que nous montrons par les zones hachurées de la figure 6.10.14. Ce qui nous donne :

base_triangle = largeur_contenu * 2
hauteur_triangle = hauteur_contenu * 2
et
largeur-boîte-graphique = base_triangle
hauteur-boîte-graphique = hauteur-triangle

Etc...

6.10.5.3 L'inter-boîte

Il est important de connaître les coordonnées d'une inter-boîte pour savoir si la place occupée par les inter-boîtes ajoutée à la place occupée par les boîtes ne dépasse pas les limites imposées sur l'espace d'affichage. Nous pouvons illustrer ceci à l'aide de la figure 6.10.15.

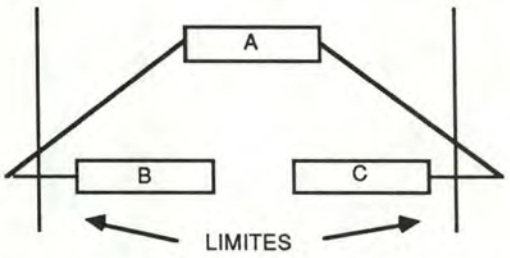


Figure 6.10.15 : importance des coordonnées de l'inter-boîte

Les boîtes A, B, C sont dans les limites. Cependant, après avoir représenté les inter-boîtes, le graphique ne respecte plus ces limites.
Une inter-boîte relie deux boîtes entre elles. La position des boîtes reliées ne dépend pas de l'inter-boîte (figure 6.10.16).

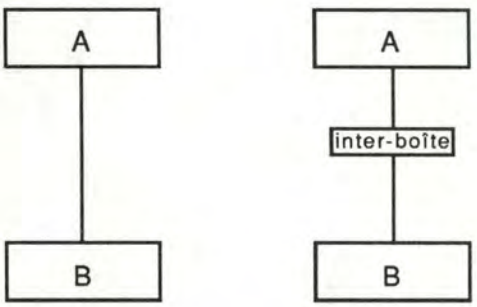


Figure 6.10.16 : indépendance de la position des boîtes reliées

Par contre comme nous le montre la figure 6.10.17, les dimensions de l'inter-boîte dépendent de la position des boîtes reliées.

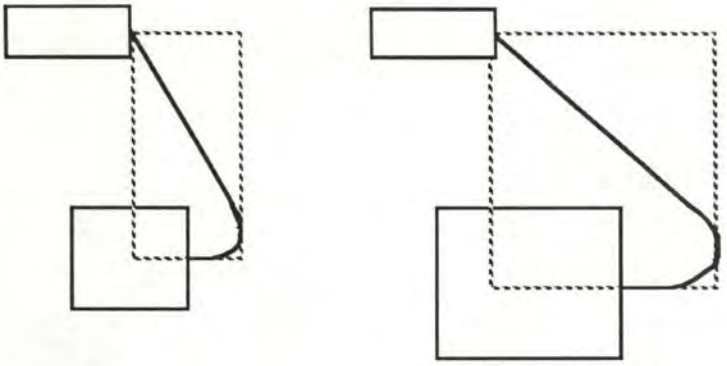


Figure 6.10.17 : dépendance des dimensions de l'inter-boîte

Dès lors, nous ne pourrons calculer les dimensions de l'inter-boîte que lorsque nous aurons attribué les coordonnées absolues aux boîtes reliées.
Nous utilisons une fonction, appelée **fonction position** qui détermine les coordonnées de la boîte qui doit contenir le concept graphique représenté par l'inter-boîte. En paramètres de cette fonction, nous introduisons les coordonnées de la boîte-mère et de la boîte-fille reliées, ainsi que les concepts graphiques représentés par ces boîtes. Cette fonction position renvoit les coordonnées, et pas seulement les dimensions de l'inter-boîte puisque cette fonction renferme l'information concernant **le point de liaison** entre la boîte et l'inter-boîte. En effet, ce point de liaison varie suivant le type de concept graphique représenté par la boîte et l'inter-boîte. En ayant seulement les dimensions de l'inter-boîte, nous ne saurions pas où la placer. Nous pouvons illustrer ceci à l'aide de la figure 6.10.18.

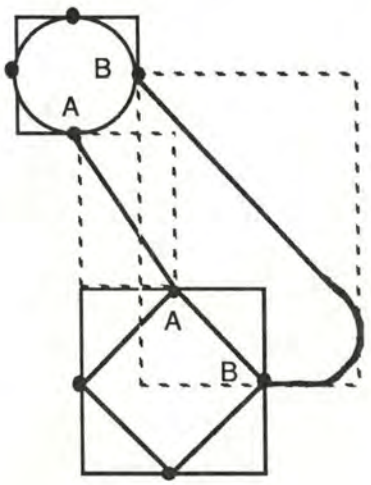


Figure 6.10.18 : variation des points de liaisons

Suivant que l'inter-boîte est une droite ou une droite-arc (cfr. 6.12), les points de liaison sont respectivement A ou B.

Description de la fonction position

Cette fonction est spécifique au concept graphique de l'inter-boîte. Dans la table des concepts graphiques (cfr. 6.10.7), elle est définie et mise en correspondance avec le concept graphique respectif. Cette fonction doit également tenir compte des concepts graphiques des

boîtes reliées pour déterminer le point de liaison. Il est possible de définir plusieurs points de liaison (ou point de référence) paramétrables sur un même concept graphique [Nanard 84]. Le paramètre exprime lequel de ces points de liaison nous devons considérer entre la boîte et l'inter-boîte. Cependant pour des raisons de facilités, nous ne considérons que des points de liaison non-paramétrables. Ces points de liaison sont fixés à priori suivant le type du concept graphique et la position de la boîte et de l'inter-boîte. Les points de liaison sont en fait, les point d'intersection du concept graphique de la boîte et du cadre qui l'entoure.
Nous donnons un exemple de fonction position pour le concept graphique droite-arc (figure 6.10.19).

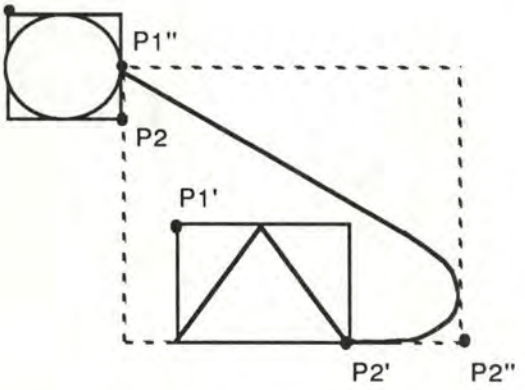


Figure 6.10.19 : fonction-positions pour une droite-angle

Exemple : *droite-angle*

inter-boîte = droite-angle

boîte-mère = cercle
⇒ point liaison - boîte-mère = (P2.x, P2.y + (P1.y-P2.y)/2))

boîte-fille = triangle
⇒ point liaison - boîte-fille = (P2.x', P2.y')

appliquer : fonction-positions-droite-arc (point liaison-boîte-mère, point liaison-boîte-fille)

résultats : P1" de coordonnées (P1".x, P1".y)
P2" de coordonnées (P2".x, P2".y)

La fonction-positions-droite-arc est la fonction spécifique au concept graphique droite-arc. Elle renvoit les points P1", P2" de la boîte nécessaire pour contenir la droite-arc allant du point liaison-boîte-mère au point liaison-boîte-fille.
Dans sa description, cette fonction doit tenir compte non seulement du concept graphique qu'elle représente, mais également de tous les concepts graphiques des boîtes qu'il est possible de relier avec cette inter-boîte, ainsi que de leur position.

Remarques

Dans tous les exemples qui précèdent, nous avons considéré le calcul des coordonnées d'une boîte lorsque celle-ci est composée d'une seule boîte. Cependant, comme nous l'avons vu dans la syntaxe de la S.I.G., les boîtes graphique et textuelle peuvent être composées de plusieurs boîtes reliées entre elles par des inter-boîtes. Le calcul des coordonnées d'une boîte composée est alors plus complexe puisqu'il s'agit de faire la somme adéquate c'est-à-dire en fonction de l'orientation des inter-boîtes, des dimensions des boîtes composantes. Ce problème n'est abordé qu'en section 6.11 puisqu'il nécessite des notions concernant le positionnement de boîtes qui sont exposées ultérieurement.

6.10.5.4 Conclusion

Le concept de boîte graphique est nécessaire. Il exprime simplement la largeur et la hauteur maximales que prend la représentation externe graphique du noeud de la S.I.G.. L'annotation n'a pas à connaître la nature du concept graphique représenté pour pouvoir établir les coordonnées de la boîte. Ce sont les fonction position et fonction dimension qui s'en chargent. Ces fonctions assurent donc l'**indépendance** de l'annotation par rapport aux concepts graphiques utilisés dans la S.I.G..

6.10.6 La désignation

Une fois que la S.I.G. est affichée à l'écran, l'utilisateur doit pouvoir y faire certaines opérations. Pour ce, il est nécessaire qu'il puisse **sélectionner** les objets affichés. Il est donc primordial de mettre en oeuvre un dispositif qui permette la **désignation** c'est-à-dire la reconnaissance dans la S.I.G. de l'objet sélectionné à l'écran à l'aide d'un clic souris par exemple. Comment reconnaître dans la S.I.G., l'objet désigné par la coordonnée écran (abscisse, ordonnée) du point de sélection?
Le concept de boîte sert également de support pour la gestion de ces entrées. Toute désignation d'un point à l'écran se traduit par la détermination dans la S.I.G., de **la boîte possesseur du point**, puis de l'entité graphique ou textuelle représentée par la boîte. Le concept de boîte se conduit donc comme un outil d'affichage et de désignation à un niveau d'abstraction élevé : le mécanisme laisse à l'application le droit d'ajuster la **granularité** de la désignation (sélection de l'entité désignée ou sélection de l'entité qui inclu cette entité désignée (cfr. 6.10.4.7 le champ POINTEUR_ARBRE_ABSTRAIT).
Nous montrons également en quoi l'annotation est **indépendante** des concepts graphiques utilisés dans la S.I.G. et définis dans la table de description des concepts graphiques.

6.10.6.1 Granularité de la désignation

Dans la représentation externe graphique, l'utilisateur voit des objets. Un objet peut être composé de graphiques ou de textes, mais pour lui, il représente une seule **entité**. La granularité de la désignation, consiste à fixer cette entité en fonction de la boîte de la S.I.G. désignée par un point écran. Cette granularité est fonction de l'application. Dans notre système, ce qui importe pour l'utilisateur, c'est de pouvoir sélectionner un objet représenté à l'écran, pour en avoir une description textuelle plus détaillée. Comme la S.I.G. est une hiérarchie de composition spatiale, l'utilisateur peut sélectionner n'importe quel objet affiché. Il peut désigner aussi bien une boîte graphique qu'une boîte élémentaire ou qu'une inter-boîte car

à chaque objet représenté, il y correspond une boîte ou noeud de la S.I.G. qui nous donne sa position écran.
Nous devons donc limiter la désignation aux boîtes de la S.I.G qui ont un **lien** avec l'arbre abstrait (ou boîtes **sélectionnables**), permettant ainsi d'effectuer une décompilation textuelle afin d'obtenir cette description textuelle plus détaillée. Sont exclues de cette catégorie, les inter-boîtes.
Lorsqu'un concept graphique entoure une zone texte, nous ne voulons pas limiter l'espace de désignation au seul texte entouré car le concept graphique et le texte forment ensemble une même entité. C'est pourquoi le lien avec l'arbre abstrait est initialisé non seulement au niveau de la boîte textuelle, mais également au niveau de la boîte composante graphique qui représente ce concept graphique. Par exemple :

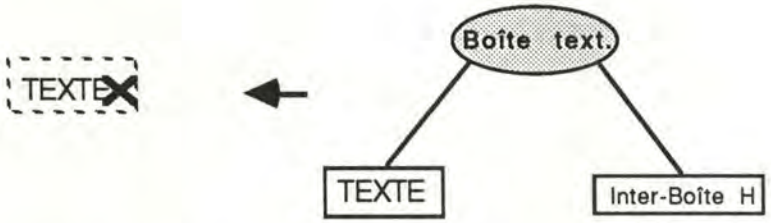


Figure 6.10.20 : boîte textuelle désignée et SIG correspondante

A la figure 6.10.20, le point écran **X** désigne la boîte textuelle représentée en pointillés. Dans la S.I.G., la boîte désignée est représentée en hachuré. Pour l'utilisateur, cette boîte est une entité. La désignation de cette entité a pour conséquence un certain traitement.

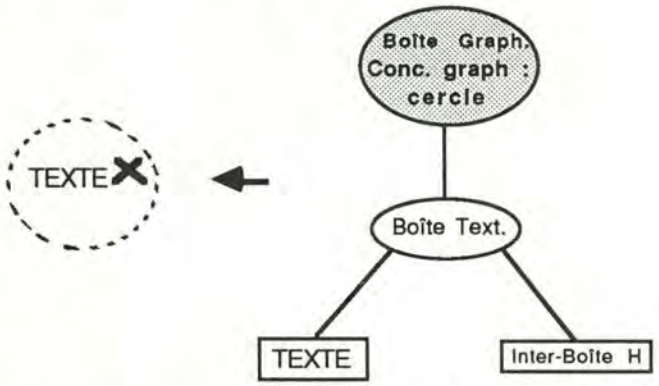


Figure 6.10.21 : boîte graphique désignée et SIG correspondante

A la figure 6.10.21, le point écran **X** désigne la boîte graphique représentée en pointillés. Pour que l'utilisateur obtienne le même résultat que celui de la figure précédente, c'est-à-dire en cliquant sur la boîte textuelle, il faut que le lien avec l'arbre abstrait soit également défini au niveau de la boîte graphique.

Nous voyons qu'en sélectionnant une entité, l'utilisateur peut désigner à la fois plusieurs boîtes de la S.I.G. Tout point écran d'une boîte composée, appartient également à l'ensemble des points écran de la boîte composante et ainsi de suite. Il y a donc plusieurs boîtes possesseur d'un même point désigné. Il est donc nécessaire de fixer la boîte que nous devons choisir.

Comme une boîte composante peut avoir un lien avec l'arbre abstrait différent de ceux de ses boîtes composées, il importe d'affiner le plus possible la granularité et de choisir par conséquent, la boîte la moins composante.

De plus, pour permettre un affichage du sous-arbre de la S.I.G. qui n'est pas encore représenté, l'utilisateur doit pouvoir aussi désigner les boîtes qui sont des boîtes abstraction alors que celles-ci n'ont pas de lien avec l'arbre abstrait.

En définitive, le processus de désignation est le suivant :

parmi les boîtes possesseur du point, nous devons choisir celles pour qui le lien avec l'arbre abstrait est défini (c'est-à-dire celles pour qui le champ POINTEUR_ARBRE_ABSTRAIT est initialisé, cfr. 6.10.4.7) ou celle qui est boîte abstraction (c'est-à-dire celle pour qui l'indicateur du champ BOITE_ABSTRACTION est positionné, cfr. 6.10.4.5). En considérant la première alternative, nous choisissons dans l'ensemble des boîtes ainsi obtenu, celle qui est la moins composante (c'est-à-dire celle pour qui le champ NUMERO_NIVEAU est le plus grand, cfr. 6.10.4.1). Nous prenons donc la boîte la moins composante parmi les boîtes possesseur du point qui ont un lien avec l'arbre abstrait.

L'algorithme de désignation se charge de déterminer la boîte possesseur du point désigné et d'indiquer les opérations à effectuer. Il peut s'agir soit d'une décompilation textuelle, ou dans le cas où la boîte désignée est une boîte abstraction, soit d'un affichage du sous-arbre non encore représenté.

6.10.6.2 Détermination de la boîte possesseur du point désigné

Lorsque nous avons une boîte textuelle, montrer l'appartenance d'un point à cette boîte est évident. En effet, il suffit de montrer que la coordonnée du point (xd, yd) est comprise dans les coordonnées de la boîte déterminées par ses deux points P1 et P2.
Nous avons que :

Si $P1.x \leq xd \leq P2.x$
et $P2.y \geq yd \geq P1.y$
alors (xd, yd) appartient à la boîte textuelle

Lorsque nous avons un boîte graphique, cela se complique un petit peu puisque nous avons des points qui appartiennent à la boîte graphique mais qui n'appartiennent pas au concept graphique représenté :

Figure 6.10.22 : boîte, concept graphique, et point désigné

Dans la mesure où nous permettons à l'utilisateur de ne désigner la boîte graphique que lorsqu'il pointe dans les limites déterminées par le concept graphique (cfr. la zone non hachurée à la

figure 6.10.22), il importe de montrer l'appartenance du point désigné à l'espace occupé par le concept graphique lui-même. Cet espace étant fonction du concept graphique représenté à l'écran, nous avons besoin d'une **fonction appartenance** qui montre l'appartenance ou pas d'un point à ce concept graphique.

La fonction appartenance

Tout comme la fonction dimension, la fonction appartenance est définie et mise en correspondance avec le concept graphique respectif dans la table des concepts graphiques (cfr. 6.10.7). Cette fonction reçoit en paramètres les coordonnées des deux points P1 et P2 de la boîte graphique qui englobe le concept graphique, ainsi que la coordonnée du point désigné. Elle renvoit une valeur booléenne, vrai ou faux suivant l'appartenance ou pas du point à l'espace occupé par le concept graphique.
Cette fonction doit en fait, à partir des coordonnées de la boîte graphique, retrouver les valeurs spécifiques au concept graphique qui permettent de déterminer l'appartenance. Par exemple le centre et le rayon d'un concept graphique cercle.
Cette fonction se base sur l'équation du lieu géométrique qui définit le concept graphique représenté. Il suffit alors, de voir si le point désigné appartient à l'ensemble des points défini par ce lieu géométrique. Pour quelques concepts graphiques, nous présentons ci-dessous, l'équation du lieu géométrique qui les définit.

Détermination des lieux géométriques

Le cercle

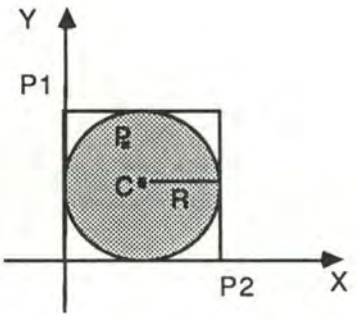


Figure 6.10.23 : le cercle

Nous savons qu'un disque D(C, R) de centre C et de rayon R est défini comme un ensemble de points x qui vérifient l'équation donnée :

distance (C, X) ≤ R.

Le rectangle

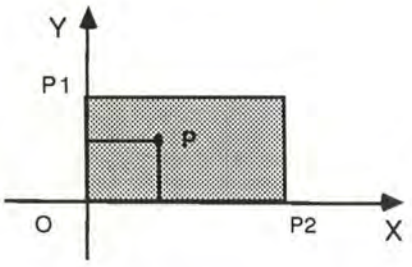


Figure 6.10.25 : le rectangle

Le rectangle est défini comme l'ensemble des points P(x, y) tels que :

$P1.x \leq P.x \leq P2.x$
et
$P1.y \leq P.y \leq P2.y$

Remarques

Nous avons vu que la granularité de la désignation est fixée par le champ POINTEUR_ARBRE_ABSTRAIT d'une boîte de la S.I.G. Ce champ représente le lien de la S.I.G. avec l'arbre abstrait; il définit les boîtes sélectionnables. Nous avons également vu que lorsque la boîte graphique et sa boîte composée forment ensemble une même entité, il est préférable que ce lien avec l'arbre abstrait soit également défini au niveau de la boîte graphique et qu'il soit le même que celui de la boîte composée. Notons cependant qu'en toute généralité, la boîte graphique peut avoir un lien avec l'arbre abstrait différent de celui de sa boîte composée, lorsque celles-ci ne forment pas ensemble une même entité. Ce qui est le cas lorsqu'une boîte graphique est composée de plusieurs boîtes qui peuvent avoir chacune un lien différent.

6.10.6.3 Conclusion

Le concept de boîte laisse à l'application le soin d'ajuster la granularité de la désignation. Par exemple si nous décidons que toute sélection de point dans une boîte graphique a pour résultat la désignation de cette boîte, il suffit de définir le lien avec l'arbre abstrait au niveau de cette boîte, sans définir de lien pour les autres boîtes composées. Lors de la désignation, le système évalue le lien d'une boîte. Si cette évaluation n'est pas "satisfaisante", il remonte dans la S.I.G. jusqu'à la boîte possesseur d'un lien. Cependant, la possibilité donnée à l'utilisateur d'ajuster la désignation est fonction de la puissance d'expression du L.D.O.R.. Cette possibilité supplémentaire est proposée en tant qu'extension du langage (cfr. la section action du langage, en section 6.5).

Le concept de boîte graphique a pour avantage d'optimiser les calculs de la désignation. En effet, la recherche de la boîte possesseur du point se fait tout d'abord en fonction des coordonnées de la boîte graphique. Ensuite, et seulement après détermination de cette boîte, la recherche se fait en fonction de l'espace occupé par le concept graphique représenté. Ainsi, la désignation ne fait appel à la fonction appartenance que si le point désigné se situe dans la boîte graphique. Sans ce concept de boîte graphique, la désignation et l'annotation en générale

seraient obligées d'utiliser la fonction appartenance pour tout concept graphique rencontré lors d'un parcours de la S.I.G. en vue de connaître l'espace occupé par le concept graphique.
L'annotation donne en paramètres à cette fonction appartenance, les coordonnées de la boîte graphique. C'est la fonction qui se charge de retrouver à partir de ces coordonnées les valeurs spécifiques au concept graphique. En ce cens, l'annotation est indépendante du concept graphique désigné.

6.10.7 La table des concepts graphiques

De manière à pouvoir ajouter facilement des concepts graphiques, ceux-ci peuvent être définis dans la table des concepts graphiques. Quelles en sont les informations nécessaires ?
Par exemple pour le concept graphique RECTANGLE il faut en première approximation le nom du concept graphique, la procédure à appeler pour afficher le concept graphique RECTANGLE. Il faut en plus pour le mécanisme de désignation, une **fonction appartenance.** Cette fonction reçoit en entrée une coordonnée déterminant une position écran (par exemple position en x et en y de la souris), ainsi que la coordonnée du coin supérieur gauche et la coordonnée du coin inférieur droit de la boîte graphique. Elle détermine à partir de ces paramètres si la coordonnée déterminant la position se trouve dans le concept graphique entouré par la boîte graphique (cfr figure 6.10.26).

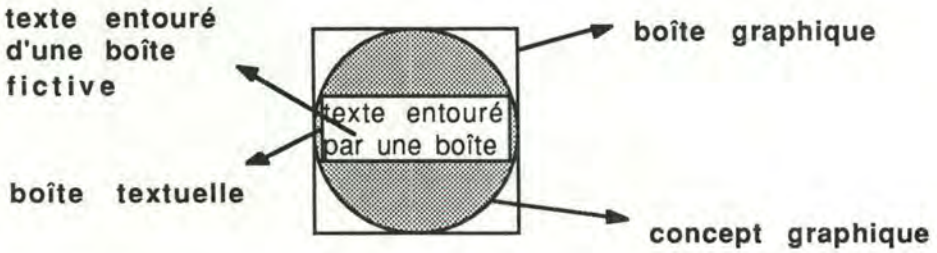


Figure 6.10.26 : les différentes boîtes

Il faut encore ajouter deux fonctions. La première recoit les deux paramètres hauteur et largeur de la boîte nécessaire pour contenir le texte qui doit être affiché dans le concept graphique. Cette fonction, appelée **fonction dimension** renvoit la taille de la boîte graphique nécessaire pour contenir le concept graphique et le texte à afficher dans le concept graphique (cfr figure 6.10.26).

La seconde, appelée **fonction position** est utilisée pour calculer la taille de la boîte nécessaire pour contenir un lien graphique entre deux objets représentés graphiquement. Cette fonction reçoit en entrée la coordonnée du coin supérieur gauche et la coordonnée du coin inférieur droit de chacune des deux boîtes (entourant les deux objets) et les deux concepts graphiques que le lien graphique doit relier. La coordonnée du coin supérieur gauche et la coordonnée du coin inférieur droit de la boîte nécessaire pour contenir le lien graphique sont renvoyés par la fonction position. La fonction position ne doit seulement être définie dans la table des concepts graphiques que pour les concepts graphiques pouvant être utilisés comme lien graphique.
Cela donnerait donc pour chaque concept graphique :

nom-concept	nom-procédure-affichage
RECTANGLE	AFFICHER_RECTANGLE

fonction appartenance	fonction dimension	fonction position
APP_RECTANGLE	DIM_RECTANGLE	POS_RECTANGLE

Le concept graphique RECTANGLE n'étant pas susceptible d'être utilisé comme lien graphique la fonction position ne doit pas être définie.
Toutes les fonctions décrites précédemment ne doivent connaître que le concept de **boîte** qui est l'**abstraction** commune partagée par toutes ces fonctions.

6.10.8 La modification incrémentale

L'utilisateur peut visualiser graphiquement une relation qui est modélisée dans l'arbre abstrait et qu'il définit à l'aide du langage de description des objets et des relations.
Pour construire sa spécification, l'utilisateur travaille sur une représentation textuelle de celle-ci. L'arbre abstrait est le support de cette représentation. Toute modification de la spécification, se répercute sur cet arbre abstrait.
Au fur et à mesure de la construction de la spécification, la relation se modifie puisqu'il y a ajout, suppression et/ou modification de composants de la spécification. Pour éviter de devoir reconstruire toute la structure intermédiaire à chaque modification d'un noeud de l'arbre abstrait, il est nécessaire de mettre en oeuvre un dispositif de modification incrémentale de la S.I.G..
Nous n'avons pas approfondi ce problème, cependant quelques remarques peuvent être apportées.
Comme signalé dans la sous-section 6.10.4, la présence du champ RELATION_REPRESENTEE dans une boîte de la S.I.G. est indispensable pour pouvoir effectuer les modifications de la S.I.G. en fonction de la relation qui y est représentée.
Il pourrait être intéressant également de répertorier les modifications de la S.I.G. suivant leur type ajout, suppression, modification de boîtes. Cette classification serait utile pour l'annotation incrémentale qui n'aurait plus à le refaire.

6.10.9 Conclusion

Le problème qui consiste à trouver une bonne S.I.G. se situe, dans notre cas, dans un débat sur la répartition de la difficulté entre l'annotation d'une part, et la description de la représentation ainsi que la décompilation d'autre part.
Le problème se complique d'autant plus lorsque la S.I.G. exprime des concepts graphiques qui peuvent se composer dans tous les sens.

Le caractère plat de la structure proposée nous oblige à tester toutes les boîtes composantes d'un niveau pour pouvoir retrouver l'information cherchée. Une structure plus hiérarchique aurait peut-être pu optimiser la recherche. Nous pouvons citer par exemple, qu'au niveau de la désignation, une structure plus hiérarchique et donc plus fidèle de la représentation externe graphique, aurait facilité la recherche de la boîte possesseur du point désigné à l'écran. Une telle disposition permettrait éventuellement de ne pas devoir tester toutes les boîtes d'un même niveau avant de trouver la boîte possesseur du point.
Au niveau de l'algorithme d'annotation, le nombre de parcours de cette S.I. est important.
Une structure, en supposant qu'elle existe et qui ne serait plus basée sur une hiérarchie de composition spatiale permettrait peut-être d'optimiser l'annotation mais compliquerait d'autant plus sa décompilation et sa description à l'aide d'un langage.

Nous avons essayé, dans la mesure du possible, de trouver un compromis.

Une généralisation de l'annotation à une représentation graphique en trois dimensions pourrait être réalisée grâce aux fonctions (fonctions appartenance, dimension et position) renfermant les secrets de représentation des graphiques. Plus généralement, il pourrait être intérressant d'étendre le L.D.O.R. à une syntaxe à trois dimensions permettant de décrire des graphiques se superposant. Cependant, ceci doit faire l'objet d'une étude ultérieure et n'est donc pas abordé dans le cadre de ce mémoire.

6.11 Annotation de la structure intermédiaire graphique

6.11.1 Introduction

Comme spécifié dans l'architecture générale de l'outil, l'**annotation graphique** consiste à passer d'une **structure intermédiaire graphique** (S.I.G.) sans notion de **positionnement** à l'écran, à une structure intermédiaire graphique avec notion de positionnement à l'écran. Cette structure intermédiaire obtenue après application de l'annotation est utilisée par l'**affichage** qui donne à l'utilisateur, en respectant le positionnement défini dans l'annotation, une **représentation externe graphique** (R.E.G.) des **objets** et de la **relation** désirée modélisée dans l'**arbre abstrait**.
Ce positionnement est important puisqu'il détermine ce que verra l'utilisateur à l'écran. Il l'est d'autant plus que nous avons comme objectif de donner à l'utilisateur une **bonne** représentation externe graphique répondant à certains critères d'ordre esthétique.
Nous devons permettre la représentation d'une relation en cours de construction. Suite à une modification incrémentale, il peut être intéressant de prévoir également une **annotation incrémentale** lorsqu'il y a modification de l'arbre abstrait entraînant une modification de la S.I.G.. Nous n'approfondirons pas ce problème complexe.
Nous présentons ensuite la **méthode** proposée pour une réalisation possible de l'annotation.

Mais avant de poursuivre plus loin cette section, nous voulons situer l'annotation graphique par rapport à l'annotation textuelle.
Dans l'architecture générale de l'outil, elles sont séparées. Il s'agit pourtant dans les deux cas, de passer d'une S.I. sans notion de positionnement à une S.I. avec notion de positionnement. Cependant, la structure physique de la S.I.G. est différente de celle de la S.I. textuelle. Néanmoins, la S.I.G. a pour but de donner une vue globale graphique d'une relation modélisée dans l'arbre abstrait. Elle nécessite donc une certaine information textuelle abrégée permettant l'interprétation de l'information graphique. A partir de la représentation externe graphique, l'utilisateur doit pouvoir obtenir la représentation externe textuelle complète de l'information. Nous devons alors établir un **lien** entre les deux structures. Celui-ci est matérialisé par les noeuds de l'arbre abstrait qui supportent l'information représentée.

6.11.2 Critères de présentation

Il est important de donner à l'utilisateur une bonne représentation externe graphique de la relation. Cette représentation doit offrir, dans la mesure du possible, une vue globale de la structure de la relation. Cependant, comme il existe une contrainte de place écran disponible, il est parfois nécessaire de choisir dans la S.I.G. ce qu'il est important de représenter à l'écran. Ceci, afin de donner cette vue globale malgré le manque de place,. De plus, cette représentation externe graphique doit être agréable à regarder. Nous devons considérer dans l'annotation de la S.I.G. des **critères de positionnement** de coordonnées respectant une certaine **esthétique**.
Nous présentons dans cette sous-section ces différentes contraintes.

6.11.2.1 Contrainte d'espacement entre les boîtes

Le système doit mettre un **espacement minimal** entre deux objets graphiques représentés à l'écran. Cet espacement minimal est fonction du type de boîte considérée dans la S.I.G.. Cette contrainte est, par défaut, prise en compte dans le système. Cependant, l'utilisateur peut l'inhiber et dire par exemple, à l'aide du L.D.O.R, que les boîtes unes telles sont collées l'une à l'autre.
Nous allons expliciter ce concept d'espacement minimal en considérant les boîtes graphiques et les boîtes textuelles de la S.I.G., ainsi que leurs représentations externes graphiques. Nous ne considérons pas les inter-boîtes, puisqu'entre celles-ci et les boîtes qu'elles relient, l'espacement minimal est toujours nul.

Boîte graphique ou textuelle non composite

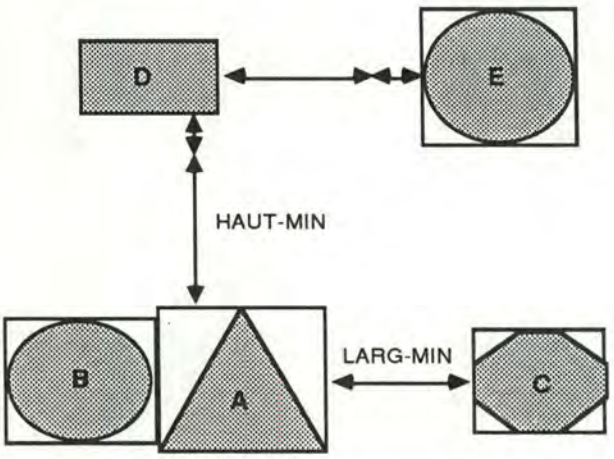


Figure 6.11.1 : espacement minimal entre boîtes non composites

Entre les boîtes reliées par des inter-boîtes EST ou OUEST :

- il est nécessaire d'avoir un espacement supérieur ou égal à la largeur minimale LARG-MIN ; les boîtes A et C de la figure 6.11.1. Par contre pour les boîtes A et B cet espacement minimal est inhibé explicitement dans la S.I.G..

Entre les boîtes reliées par des inter-boîtes SUD ou NORD :

- il est nécessaire d'avoir un espacement supérieur ou égal à la hauteur minimale HAUT-MIN comme les boîtes A et D ou A et E.

Boîte graphique composite

Exprimer l'espacement minimal entre une boîte composée et sa composante, est un peu plus compliqué. En effet, nous ne voulons pas de points d'intersection entre la représentation externe du concept graphique ou du texte composant et la représentation externe du concept graphique composé. La figure 6.11.2 montre ce que nous voulons éviter.

Figure 6.11.2 : représentations avec points d'intersection

C'est pourquoi entre la boîte graphique ou textuelle composante et la boîte graphique composée, il doit y avoir également un espacement minimal. Ce que nous représentons dans la figure 6.11.3 par une flèche à double sens.

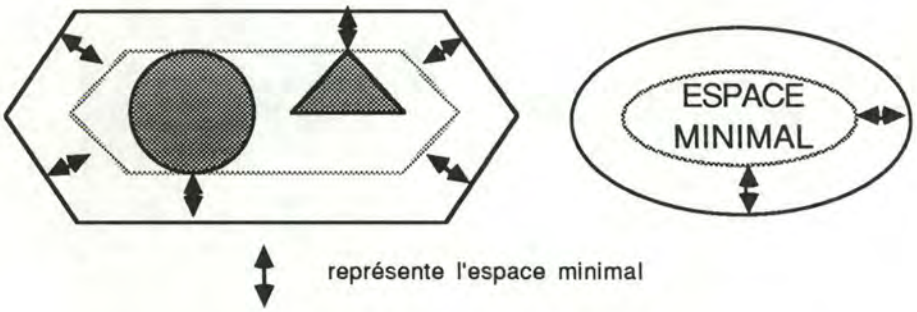


Figure 6.11.3 : représentations sans points d'intersection

Nous avons vu à la section précédente (cfr. 6.10), que nous pouvions en termes de coordonnées du cadre qui l'entoure, exprimer l'espace occupé par une boîte. Une idée de réalisation de ce critère d'espacement minimal est de considérer la boîte composante plus grande qu'elle ne l'est réellement. Dans le calcul des coordonnées de la boîte composante, nous donnons donc à la fonction dimension les coordonnées de la boîte composée, majorées de la marge que représente cet espacement minimal. Nous pouvons présenter ce calcul à l'aide de la figure 6.11.4 dont les traits en pointillés représentent les boîtes telles qu'elles sont considérées par la boîte composante et en traits continus les boîtes réelles.

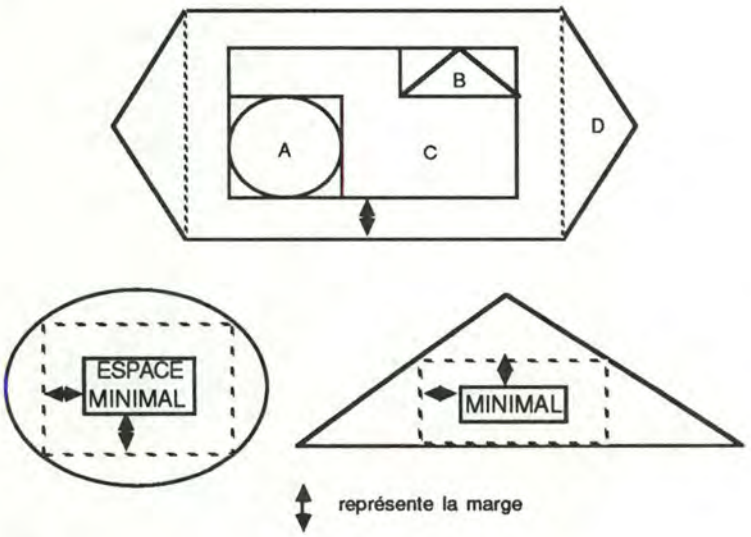


Figure 6.11.4 : espacement minimal et boîtes fictives

Nous donnons à la fonction dimension correspondante au concept graphique de la boîte D, les coordonnées de la boîte C majorées d'une marge d'espacement, c'est-à-dire les coordonnées de la boîte fictive représentée en pointillés.

Une autre solution est de laisser à l'utilisateur le soin d'expliciter dans le code de la fonction dimension cet espacement minimal qui n'est donc par fourni par défaut par le système. Cette solution permet, si l'utilisateur ne spécifie rien dans la fonction dimension, d'obtenir des représentations graphiques qui ne respectent pas ce critère. Cependant il incombe à l'utilisateur de le spécifier.

Nous pouvons remarquer que l'espace minimal qui doit exister entre une fenêtre à l'écran et la représentation externe graphique dessinée est un cas particulier de l'espacement minimal entre boîtes composites, où la boîte composante est le graphique affiché et la boîte composée est la fenêtre.

6.11.2.2 Contrainte sur l'espace d'affichage

L'espace d'affichage disponible est limité en largeur et en hauteur. L'annotation accepte en paramètre la fenêtre dans laquelle doit s'afficher la S.I.G.. L'annotation doit poser des coordonnées sur les boîtes de la S.I.G. en assurant le mieux possible une bonne représentation externe graphique dans les limites en largeur et en hauteur de la fenêtre donnée. Lorsque la fenêtre d'affichage n'est pas donnée en paramètre, l'annotation met des coordonnées sur les boîtes en fonction d'une fenêtre d'affichage dont la taille standard maximale et la position écran sont fixées à priori. L'annotation peut diminuer mais pas augmenter la taille de cette fenêtre au gré de ses besoins.
Elle a pour fonction d'occuper de la meilleure manière l'espace d'affichage disponible. Autrement dit, lorsque cet espace d'affichage est suffisant il faut répartir la représentation externe graphique sur tout l'espace pour diminuer l'espace inoccupé. A l'inverse, lorsque l'espace d'affichage n'est pas suffisant, il importe de représenter le plus d'information possible endéans les limites de la fenêtre et en respectant certains critères esthétiques. Il est donc nécessaire que l'annotation puisse jouer avec les polices de caractères pour diminuer ou augmenter la taille du graphique.

Elle doit disposer également d'un dispositif qui permette **l'élision** [Coutaz 85] sur un S.I.G.. Avec l'élision, l'utilisateur a l'avantage de pouvoir éclater la représentation externe graphique d'une relation dans plusieurs fenêtres. A l'inverse, le **défilement** ne convient pas ici puisqu'il ne permet pas un affichage simultané de plusieurs parties d'une même représentation externe graphique d'une relation. Vu notre objectif premier qui est de fournir une vue globale de la structure de la relation, il est nécessaire que l'utilisateur puisse, même à l'aide de plusieurs fenêtres, voir cette structure globale.

Pour mettre en oeuvre ce dispositif d'élision, nous utilisons une **boîte abstraction** comparable au concept du noeud abstraction [Nerson 85] qui permet de faire abstraction de la représentation externe d'une partie du graphique lorsque l'espace d'affichage disponible n'est pas suffisant pour un affichage du graphique complet. Le concept de boîte permet cette distribution par des manipulations d'arbres et par des associations de sous-arbres à diverses fenêtres.

6.11.2.3 Le concept d'élision (ou holophraste)

L'élision est combinée à un objectif de lisibilité du graphique. En effet, il est important de faire une "bonne" abstraction qui ne cache pas trop à l'utilisateur et qui surtout ne le met pas dans une situation ambigüe. Nous devons essayer, dans la mesure du possible, que l'utilisateur devine la structure cachée par la boîte abstraction. Aussi devons-nous définir la portée de cette boîte pour déterminer les autres boîtes qu'elle peut couvrir.

Portée de la boîte abstraction

Plusieurs cas de figures peuvent se produire suivant lesquels la portée de la boîte abstraction varie.
1) Supposons une boîte qui dépasse en **largeur** la limite imposée.
Représenter cette boîte par une boîte abstraction en diminue les dimensions. Nous la représentons par une boîte contenant trois points (cfr. 6.11.5).

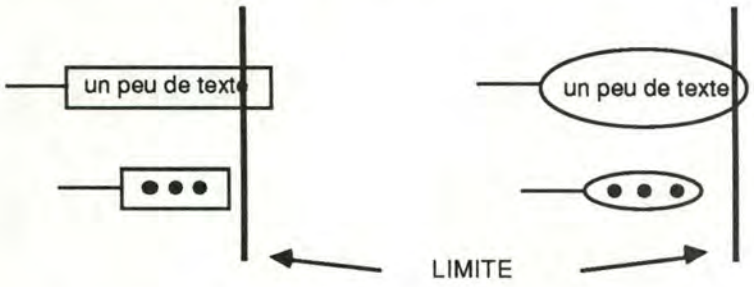


Figure 6.11.5 : la boîte abstraction diminue les dimensions de la boîte

Portée de la boîte abstraction : la boîte.
La boîte abstraction ne couvre qu'elle même, étant donné qu'elle n'a pas de boîtes à laquelle elle est reliée et qui à cause d'elle ne pourraient être représentées.

2) Généralisons le cas de figure précédent et considérons les boîtes composites.
La boîte composite peut être trop grande, ce qui la rend non représentable ainsi que ses boîtes composantes.

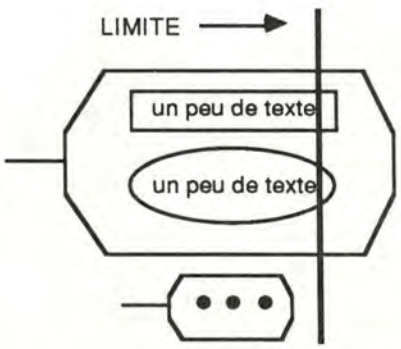


Figure 6.11.6 : généralisation aux boîtes composites

Portée de la boîte abstraction : les boîtes composantes.

3) Supposons que nous ayons une structure du genre structure PC de SACSO et que nous n'ayons pas assez de place pour dessiner les "boîtes filles". Nous devons alors représenter la "boîte mère" par la boîte abstraction.

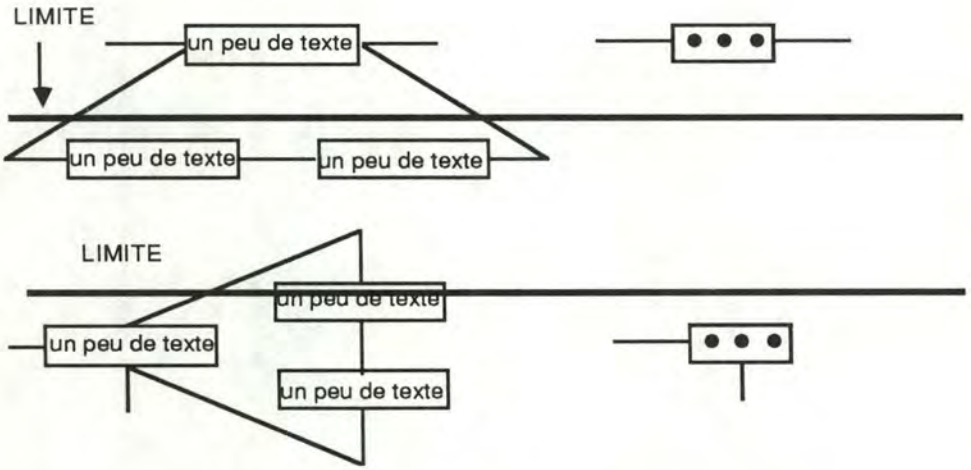


Figure 6.11.7 : impossibilité de dessiner les boîtes-filles

Portée da la boîte abtsraction : la boîte mère et les boîtes filles.

4) Nous présentons ici un exemple dans lequel il est possible de représenter les boîtes mais pas les inter-boîtes. La structure ne pouvant pas être représentée convenablement, nous faisons abstraction de la boîte mère.

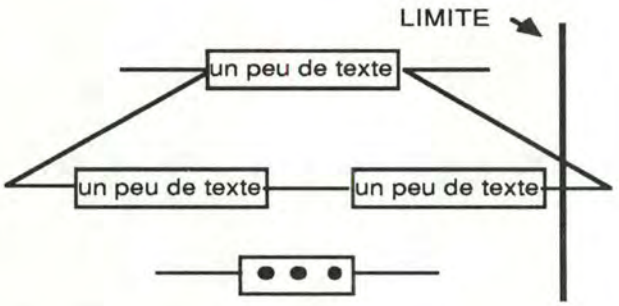


Figure 6.11.8 : impossibilité de dessiner les inter-boîtes

Portée de la boîte abtsraction : la boîte mère et les boîtes filles.

Remarques

Nous avons montré que la portée de la boîte abstraction peut être différente. Cette abstraction peut être représentée à l'écran par une boîte dont le contenu est la suite de caractères "...". Ce contenu peut être plus compliqué et intégrer par exemple le type du noeud de la S.I.G. qui est non représenté ainsi que le nombre de boîtes qui sont cachées. Cette boîte abstraction peut être textuelle ou graphique et représenter le même concept graphique que la boîte non représentée.

Remarquons également qu'avant de faire abstraction d'une boîte, nous pouvons essayer de faire de la place pour permettre une représentation complète de la boîte. Dans la figure 6.11.7, le processus d'abstraction doit voir tout d'abord, s'il n'est pas possible de représenter les boîtes filles par des boîtes abstractions. Ce qui permettrait de représenter complètement la boîte mère ainsi que la structure qui la lie aux boîtes filles. En effet, représenter les boîtes filles par trois points, diminue leur dimension (figure 6.11.9).

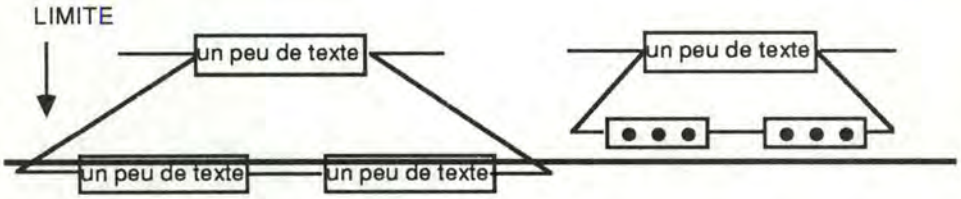


Figure 6.11.9 : récupération de la place

Il en va de même pour les figures 6.11.8 et 6.11.6. dans lesquelles, le fait de représenter les boîtes composantes par trois points diminue la dimension de la boîte composite.
Nous avons vu à l'aide de ces figures, que pour offrir une bonne élision, il est nécessaire de se définir au niveau de la S.I.G., les notions de boîtes mères et boîtes filles. Ce que nous définissons ci-après comme les concepts d'**unité**, de boîte **mère-unité** et **fille-unité**.

Concept d'unité

Une boîte définit une unité dont elle est la mère-unité. Une boîte peut également appartenir à une autre unité dont elle est la fille-unité. Une boîte B est fille-unité d'une unité U si l'une des conditions suivantes est remplie :

- la boîte B est suivante de la boîte mère-unité de l'unité U, dans le graphe de la représentation externe,
- ou la boîte B n'est pas suivante de la boîte mère-unité de l'unité U, mais elle appartient à un chemin joignant deux boîtes suivantes de la boîte mère-unité de l'unité U, toujours dans le graphe de la représentation externe.

Ce concept est important dans la mesure où il permet de déterminer si une boîte qui n'est pas reliée à la boîte mère-unité, fait quand même partie de l'unité. Supposons que nous n'ayons pas ce concept. Faire abstraction d'une telle boîte consiste normalement à faire abstraction des autres boîtes qui sont extrémités terminales d'une inter-boîte dont la boîte abstraction est extrémité initiale.

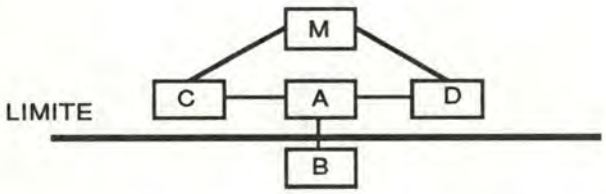


Figure 6.11.10 : impossibilité de dessiner B

Dans la figure 6.11.10, rien ne nous permet, au niveau de la boîte A, de distinguer les boîtes B et D en dehors du fait qu'elles ne sont pas reliées à A par les mêmes inter-boîtes. Ainsi, n'ayant pas assez de place pour dessiner la boîte B, nous remontons jusqu'à l'extrémité initiale de l'inter-boîte en l'occurence A, pour en faire une boîte abstraction. Comme D est également extrémité terminale de A, elle ne peut être dessinée. Cela nous donne une mauvaise abstraction car elle cache la structure (figure 6.11.11).

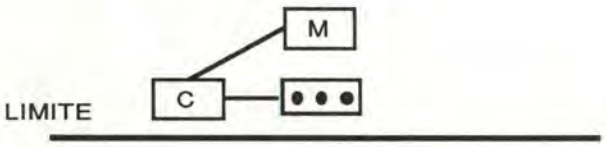


Figure 6.11.11 : la structure est cachée

Par contre, avec le concept d'unité, nous pouvons "compléter" la représentation de l'unité dont la boîte A est la fille, en l'occurence l'unité dont la boîte M est la mère (figure 6.11.12).

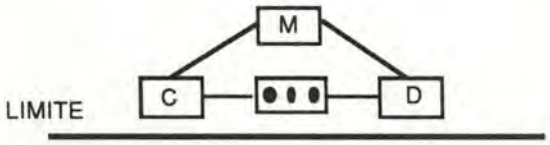


Figure 6.11.12 : la structure n'est pas cachée

Il existe une inter-boîte entre A et D mais cependant D n'est pas une boîte fille-unité de A de sorte que nous essayons de complèter la représentation de l'unité M. En effet, C, A et D sont des boîtes filles-unité de M. Ainsi, parmi les boîtes reliées à A avec une inter-boîte dont A est extrémité initiale, nous avons des boîtes qui sont filles-unité de A et d'autres qui sont filles-unité de M. Nous pouvons, en guise de conclusion, donner la règle suivante :

- lorsque nous faisons abstraction d'une boîte, cette abstraction ne doit couvrir que les boîtes qui sont filles-unité de la boîte considérée. Cette définition est bien sûr récursive et s'applique aux boîtes filles-unités des boîtes filles-unité et ainsi de suite.

Cette notion d'unité est représentée, au niveau de la S.I.G., par une information supplémentaire dans chaque boîte de la S.I.G. (cfr. le champ IDENTIFIANT_BOITE_MERE_UNITE de la boîte en section 6.10.4.3) et qui renseigne la mère de l'unité dont la boîte fait partie. Cette information doit être calculée au préalable, comme expliqué ci-après.

Calcul des unités

Le principe de calcul de l'unité consiste à déterminer un chemin dans la S.I.G., qui rejoint deux boîtes filles-unité qui sont extrémités terminales d'une inter-boîte dont la boîte mère-unité est extrémité initiale. Ce principe se base sur le fait que les inter-boîtes expriment les arcs orientés d'un graphe. Ce graphe n'est autre que la représentation externe graphique. La S.I.G. quant à elle est une structure arborescente. L'orientation de l'arc est donnée par l'orientation de l'inter-boîte. C'est pourquoi nous parlons d'extrémité initiale et terminale de celle-ci.
Dans l'exemple présenté ci-avant (fig. 6.11.12), nous avons :

- C est fille-unité de M car il existe une inter-boîte de M à C
- D est fille-unité de M car il existe une inter-boîte de M à D
- il existe un chemin allant de C à D, qui fait que toutes les boîtes sur le chemin sont filles-unité de M, en l'occurence la boîte C qui n'est pas fille-unité de A.

Voici un exemple qui montre comment procéder lorsque nous avons une boîte mère-unité de différentes unités (figure 6.11.13).

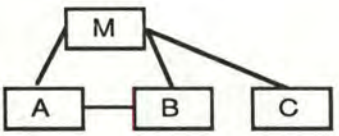


Figure 6.11.13 : boîte mère-unité de plusieurs unités

- A est fille-unité de M
- B est fille-unité de M
- C est fille-unité de M
- il n'existe pas de chemin allant de A à C, ou de C à A
- il n'existe pas de chemin allant de B à C, ou de C à B
- il n'existe pas de chemin allant de B à A
- il existe un chemin allant de A à B

En toute généralité, une boîte peut être fille-unité de plusieurs unités différentes (figure 6.11.14). Auquel cas, plusieurs chemins différents peuvent rejoindre deux mêmes boîtes. Dans ce cas, le chemin le plus court définit l'unité.

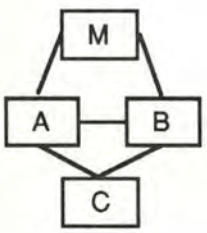


Figure 6.11.14 : une boîte fille-unité de plusieurs mères-unité

- il existe un chemin allant de A à C et passant par B. Ce qui fait que normalement B doit être fille-unité de A. Cependant, il existe un chemin plus court représenté par l'inter-boîte entre A et C. La boîte B n'est donc pas fille-unité de A mais bien de M. Par contre C est fille-unité de A et de B. Au cas où C ne peut pas être représenté, les boîtes A et B seront des boîtes abstraction et pas seulement une des deux.

Classement des inter-boîtes

La détermination des unités requiert un certain classement des inter-boîtes qui permet de diminuer le nombre de parcours dans la S.I.G..

Nous pouvons par exemple classer par boîte toutes les inter-boîtes pour lesquelles cette boîte est extrémité terminale ou extrémité initiale.

Nous avons donc vu que l'annotation peut jouer avec les polices de caractères ou avec les noeuds abstraction pour permettre un affichage de la S.I.G. dans les limites d'une fenêtre donnée en paramètre ou d'une fenêtre de dimensions standards fixées à priori. Mais étant donnée cette fenêtre, quels sont les critères que doit respecter l'annotation et qui lui assurent la **lisibilité** du graphique représenté?

6.11.2.4 Contraintes d'ordre esthétique

Nous présentons les différentes contraintes, ainsi que leur formalisation.

Répartition homogène des boîtes

La répartition homogène des boîtes se base sur les notions de mère-unité et de fille-unité afin d'assurer un graphique équilibré. Cet équilibrage consiste essentielement à **centrer** les boîtes filles-unité par rapport aux boîtes mères-unité et vice-versa. Remarquons que lorsque la boîte mère-unité unique n'a qu'une seule fille-unité, le centrage assure la **verticalité** et l'**horizontalité** du graphique. Remarquons enfin que le critère de centrage peut ne pas être respecté lorsque l'utilisateur veut explicitement que le graphique ne soit pas équilibré.

Contraintes

<C1> Une boîte mère-unité est placée au milieu de ses boîtes filles-unité

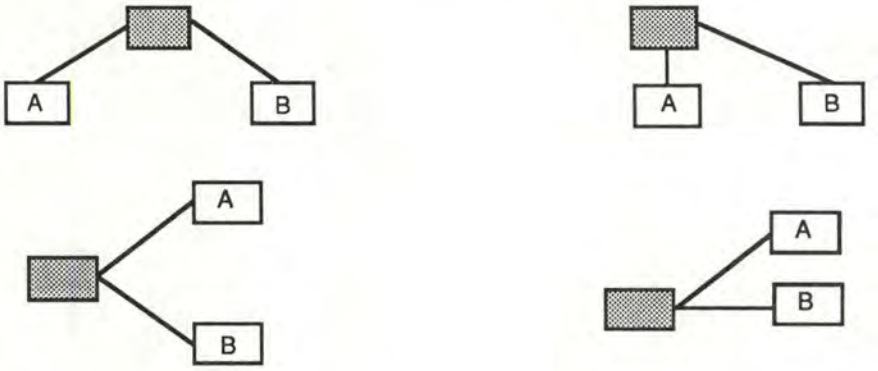


Figure 6.11.15 : <C1> respectée *<C1>non respectée*

<C2> Une boîte fille-unité est placée au milieu de ses boîtes mères-unité

Figure 6.11.16 : <C2> respectée *<C2> non respectée*

<C3> Les boîtes filles-unité sont centrées entre elles par rapport à une même ligne de base.

Cette contrainte est nécessaire vu que les boîtes peuvent avoir des dimensions différentes.

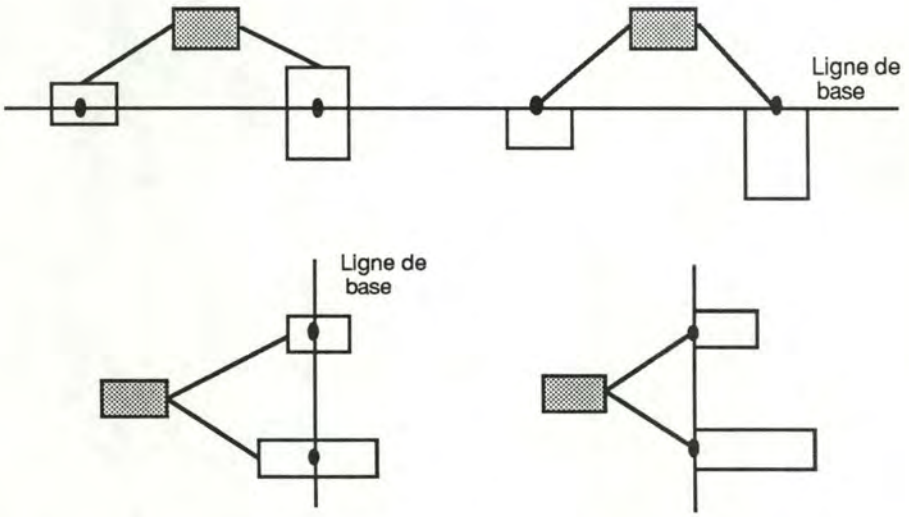


Figure 6.11.17 : <C3>respectée *<C3> non respectée*

Remarques

1) Sur un strict point de vue esthétique, nous ne pouvons imposer la contrainte <C3>. En effet, dans certains graphiques (fig. 6.11.18) les exemples de dessins qui ne respectent pas <C3> peuvent être plus esthétiques que ceux qui la respectent, mais sur d'autres graphiques, cela peut être l'inverse (cfr. 6.11 19). Nous pourrions alors proposer une nouvelle contrainte <C4> définie comme étant la contrainte <C3> dans laquelle la ligne de base ne passe pas par le centre des boîtes (cfr. les exemples de dessins qui ne respectent pas <C3> dans la figure 6.11.17). Cependant, une telle contrainte serait moins homogène. Suivant l'orientation des boîtes filles-unité (SUD ou EST par exemple), le point de référence (cfr. le point noirci dans la figure 6.11.17) qui permet le centrage, est différent.

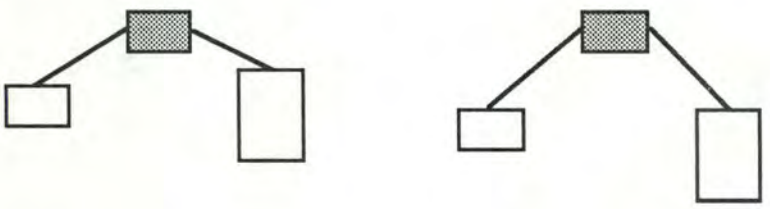


Figure 6.11.18 : <C3> respectée *<C3> non respectée*

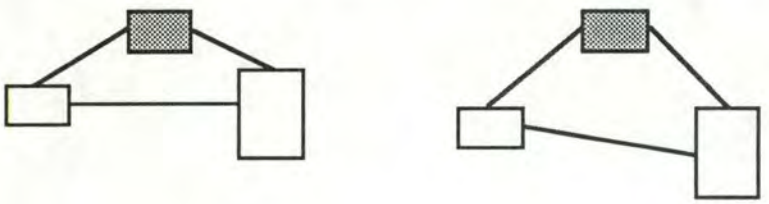


Figure 6.11.19 : < C3> respectée *<C3> non respectée*

2) La verticalité et l'horizontalité du dessin sont assurées par le centrage. Ce sont des cas particuliers où la boîte mère-unité unique n'a qu'une seule boîte fille unité.

Figure 6.11.19 : verticalité et horizontalité

3) Les contraintes <C1>, <C2>, <C3> peuvent être combinées entre elles suivant que les boîtes filles-unité n'ont pas toutes la même orientation. Nous voyons dans le figure 6.11.20, que la contrainte <C1> s'applique aux boîtes A et B, tandis la contrainte <C3> s'applique aux boîtes C et B.

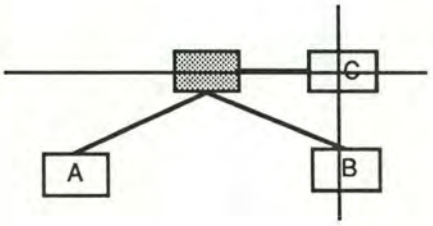


Figure 6.11.20 : combinaison des contraintes

Aussi, nous voyons que l'applicabilité des contraintes est fonction de l'orientation de la boîte considérée.

4) Les contraintes <C1> et <C2> sont assurées par le système, lorsque l'utilisateur ne les inhibe pas. Dans la figure 6.11.15, l'utilisateur a défini les boîtes filles-unité comme étant situées au SUD de la boîte mère-unité. Il aurait pu également positionner toutes les boîtes filles-unité au NORD-EST ou SUD-EST de la boîte mère-unité (figure 6.11.21).

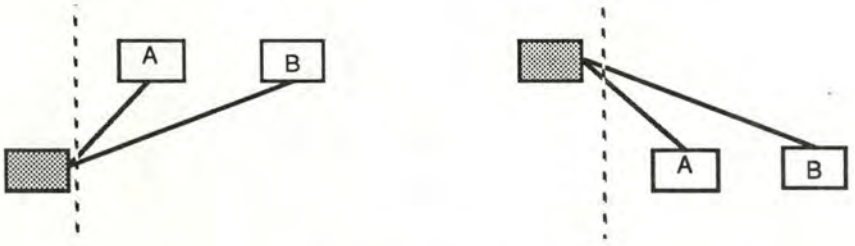


Figure 6.11.21 : boîtes filles-unité au NORD-EST ou au SUD-EST

Cette dernière remarque se base sur la notion de **division de l'espace d'affichage** suivant les points cardinaux, notion qui est définie et développée dans une section ultérieure.

Formalisation des contraintes

Nous devons, pour assurer un graphique équilibré, tenir compte des coordonnées verticales (ou ordonnées) et des coordonnées horizontales (ou abscisses) des boîtes.
Ces contraintes doivent être exprimées par rapport à un point de référence identique pour toutes les boîtes. Ce point de référence doit être indépendant de la dimension des boîtes qui varie d'une boîte à l'autre. Nous choisissons d'exprimer les contraintes par rapport aux coordonnées d'un point **centre de la boîte**. Nous pouvons supposer, pour le moment, que ce point exprime les coordonnées d'une boîte de dimension nulle. Nous présentons ultérieurement ce que représente réellement ce point centre et les problèmes qu'il engendre.

Contrainte <C1>

1) Centrage horizontal

$$\text{boîte-mère-unité.x} = \left(\Sigma_{\text{boîte-fille-unite}} \text{ boîte-fille-unité.x} \right) / \text{nbre-boîte-fille-unité}$$

2) Centrage vertical

$$\text{boîte-mère-unité.x} = \left(\Sigma_{\text{boîte-fille-unite}} \text{ boîte-fille-unité.x} \right) / \text{nbre-boîte-fille-unité}$$

Dans les deux cas, la somme se fait sur l'ensemble des boîtes filles-unité pour lesquelles la contrainte est applicable.

Cas particuliers (verticalité et horizontalité)

nbre-boîte-fille-unité = 1 et nbre-boîte-mère-unité =1

Contrainte <C2>

Nous devons ici tenir compte des dimensions de la boîte :

P1.x = point-centre.x - (largeur-boîte/2)
P2.x = point-centre.x + (largeur-boîte/2)
P1.y = point-centre.y + (hauteur-boîte/2)
P2.y = point-centre.y - (hauteur-boîte/2)

Le nombre de boîtes

Ce critère tente d'exprimer qu'un graphique avec trop de boîtes sur la largeur et avec trop de boîtes de hauteur différente n'exprime plus de façon claire, une vue globale de la structure d'une relation parce que le graphique est trop chargé. Ainsi pour la représentation externe graphique d'une même information, il semble plus esthétique d'avoir un plus grand nombre de fenêtres, chacune d'elles un peu moins remplies qu'un nombre de fenêtre plus restreint, chacune d'elles étant un peu plus remplies.

Il ne sagit pas de changer la structure de la S.I.G. mais simplement de faire le choix d'une représentation externe graphique éclatée en un nombre de fenêtres plus grand que ne le ferait une annotation graphique qui ne tiendrait pas compte de ce critère.

6.11.3 Mise en oeuvre du concept d'élision

La structure physique de la S.I.G., n'est pas une image fidèle de ce qui est représenté à l'écran. Cependant, pour chaque concept graphique ou textuel qui est affiché, il y correspond une boîte dans la S.I.G.. Cette boîte contient sa position écran.
Pour mettre en oeuvre un mécanisme d'élision, matérialisé par une boîte abstraction affichée à l'écran, nous devons définir ce à quoi correspond cette boîte au niveau de la S.I.G..
La solution envisagée consiste à considérer la boîte abstraction comme une boîte également représentée dans la S.I.G. tout comme les autres boîtes. Lorsqu'il est nécessaire de faire abstraction d'une boîte, le module d'annotation crée une boîte abstraction qu'il insère dans la S.I.G.. Cette boîte insérée "remplace" la boîte dont elle fait abstraction.
Cette boîte peut nécessiter un traitement particulier lors de sa désignation (par exemple un affichage de la partie non représentée), aussi il est nécessaire de pouvoir l'identifier comme étant une boîte abstraction. C'est l'objet du composant indicateur du champ BOITE-ABSTRACTION de la boîte.
Nous présentons le principe qu'il est possible de suivre pour l'insertion de cette boîte dans la S.I.G..

Insertion de la boîte abstraction dans la S.I.G.

Une première solution consiste à créer une autre S.I.G. dans laquelle nous ne représentons que les boîtes qui sont affichées et donc parmi celles-ci la boîte abstraction. La figure 6.11.22 présente la solution :

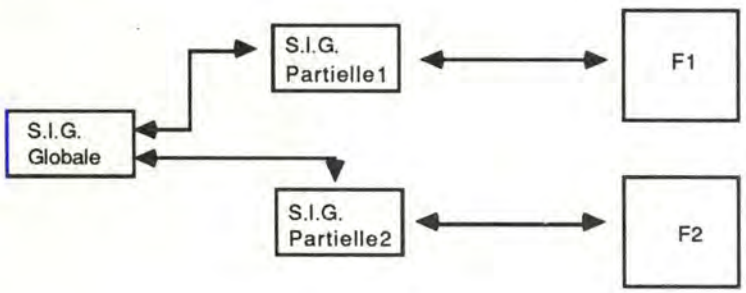


Figure 6.11.22 : première solution

La S.I.G. partielle ne reprend de la S.I.G. globale que ce qui est affiché à l'écran. Nous n'avons donc qu'une seule S.I.G. globale et plusieurs S.I.G. partielles, chacune correspondant à une partie de la S.I.G. globale affichée dans une fenêtre. Nous devons garder la correspondance entre les S.I.G. globale et partielles, pour permettre à l'utilisateur de visualiser la partie non encore affichée de la S.I.G. globale. Cette correspondance est représentée dans la figure 6.11.22 par une flèche à double sens.
Cette solution présente l'inconvénient d'augmenter considérablement le nombre de structures à manipuler.

La deuxième solution consiste simplement à créer une boîte abstraction au lieu d'une nouvelle S.I.G.. Cette boîte créée fait la correspondance entre la boîte dont elle fait l'abstraction, et la fenêtre dans laquelle la boîte abstraction ainsi que la S.I.G sont affichées. Nous n'avons qu'une seule S.I.G. et plusieurs boîtes abstraction supplémentaires lorsque la représentation est éclatée dans plusieurs fenêtres.
Le composant pointeur-boîte-abstraction du champ BOITE-ABSTRACTION de la boîte permet à partir de la boîte de la S.I.G., de retrouver la boîte abstraction créée. Celle-ci contient le pointeur vers la fenêtre d'affichage.
Une illustration de cette solution est donnée à la figure 6.11.23.

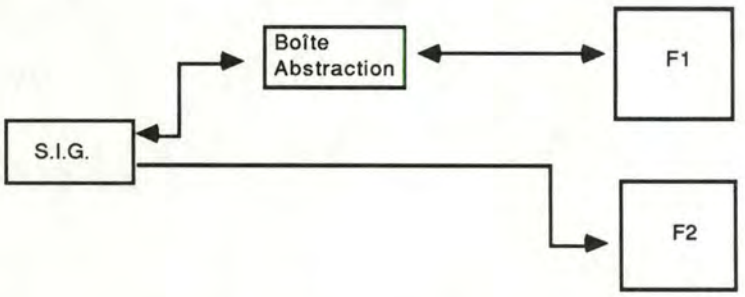


Figure 6.11.23: deuxième solution

6.11.4 Méthodologie proposée

6.11.4.1 Introduction

Nous proposons en premier lieu une **architecture générale** pour la réalisation du module d'annotation, et ensuite les différents **algorithmes** identifiés dans cette architecture. Nous présentons ensuite une proposition de **méthode** pour la réalisation de l'algorithme d'annotation. Cette méthode s'est inspirée de différentes méthodes existantes en la matière.

6.11.4.2 Architecture générale du système

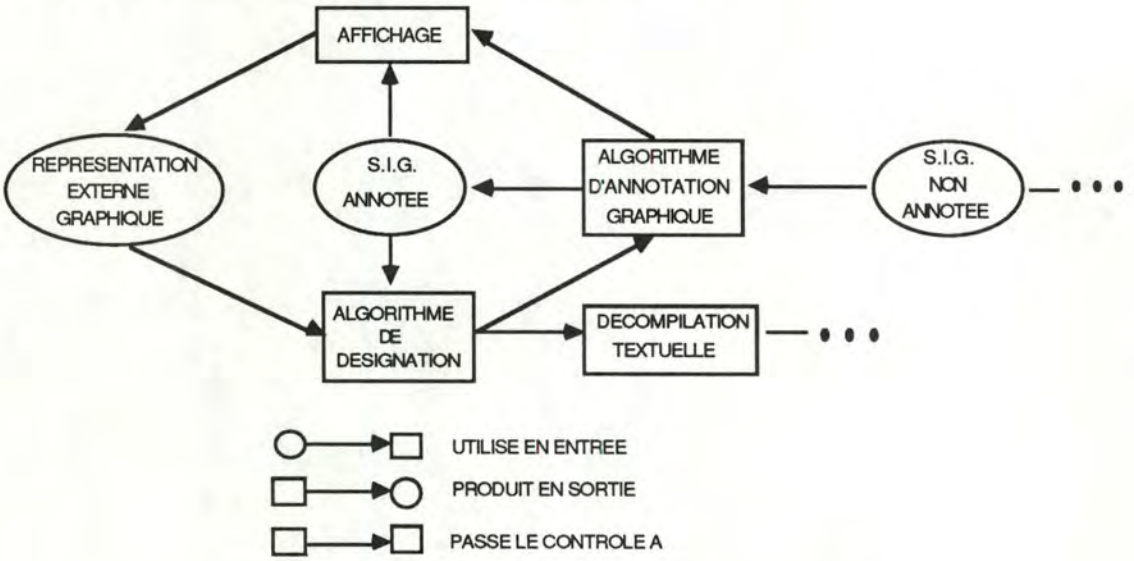


Figure 6.11.24 : architecture générale de l'annotation

L'utilisateur désigne à l'écran, par exemple au moyen d'une souris, les objets sur lesquels il désire opérer. L'**algorithme de désignation** a pour rôle de faire la correspondance entre la représentation externe et la structure interne qui la supporte. Suivant l'objet désigné, l'opération effectuée est soit une **décompilation textuelle** ou une **annotation graphique**. L'**algorithme d'annotation graphique** a pour rôle de produire la **S.I.G. annotée** et de tenir à jour la représentation externe graphique par l'intermédiaire de l'**affichage**.

6.11.4.3 Algorithmes de désignation

1) Algorithme principal

Spécification

L'exécution de cet algorithme a pour effet de fournir en sortie l'identificateur de l'opération ainsi que ses arguments correspondant à l'entité sélectionnée par l'utilisateur. Pour déterminer cette entité, l'algorithme doit tenir compte de la granularité de la désignation définie en 6.10.6.
Les identificateurs d'opération possible sont soit une annotation graphique, soit une décompilation textuelle, soit en toute généralité, d'autres opérations comme déplacer, agrandir, rendre invisible, etc..

- *arguments* :
  point_désigné : une position écran
  fenêtre : l'identificateur d'une fenêtre

- *pré-condition* :
  fenêtre identifie une fenêtre existante.

- *résultats* :
  opération : identificateur d'une opération
  arguments : la liste des arguments de l'opération

- *post-conditions* :
  opération identifie une opération existante. Les identificateurs d'opération possible sont les suivants :
  - annotation_graphique : dans ce cas, les arguments sont l'identificateur de la boîte désignée, l'identificateur de la S.I.G. désignée, et la fenêtre d'affichage,
  - décompilation_textuelle : dans ce cas, les arguments sont le lien avec l'arbre abstrait de la boîte désignée,
  - autres : il peut en effet y avoir d'autres opérations,
  - vide : dans ce cas, il n'y a pas d'opération et les arguments sont sans importance.

Réalisation abstraite

Idée intuitive

L'idée consiste tout d'abord à faire la recherche de la boîte possesseur du point désigné compte tenu de la granularité. Suivant les caractéristiques de la boîte désignée (boîte-abstraction ou pas, ...) l'opération à effectuer est différente.

DESIGNATION (point_désigné, fenêtre)

```
début
    opération, arguments := VIDE
    SIG_désignée := {chercher la S.I.G. correspondant à la fenêtre fenêtre}
    Si  SIG_désignée existe
        Alors
            boîte_désignée := DETERMINER_BOITE_DESIGNEE_POINT (SIG_désignée, point_désignée)
            Casoù boîte_désignée
                    =VIDE :
                        ne rien faire
                    =boîte abstraction :
                        fenêtre_abstraction := {créer une nouvelle fenêtre d'afichage}
                        opération := annotation_graphique
                        arguments := identifiant-boîte (boîte_désignée), SIG_désignée, fenêtre_abstraction
                    ={autres} :
                        opération :=  décompilation_textuelle
                        arguments :=pointeur-arbre abstrait (boîte_désignée)
                fincasoù
    finsi
    retourner (opération, arguments)
fin
```

2) Déterminer la boîte désignée par un point écran

Spécification

Cette fonction consiste à retrouver la boîte de la S.I.G. désignée par un point écran.

- *arguments* :
  SIG_désignée : identificateur d'une S.I.G.
  point_désigné : position écran

- *pré-condition :*
  SIG_désignée identifie une S.I.G. existante

- *résultat :*
  boîte_désignée : identificateur d'une boîte de la S.I.G.

- *post-conditions :*
  SI l'identificateur boîte_désignée de la boîte désignée est VIDE,
  alors il n'existe pas de boîte de la S.I.G. identifiée par SIG_désignée respectant la définition de la granularité de la désignation et désignée par la position écran point_désigné,

sinon boîte_désignée identifie la boîte de la S.I.G. identifiée par SIG_désignée respectant la définition de la granularité de la désignation et désignée par la position écran point_désigné. Le résultat boîte_désignée identifie soit :

- une boîte abstraction : l'indicateur du champ BOITE_ABSTRACTION de la boîte est alors positionné,
- une boîte sélectionnable : le champ POINTEUR_ARBRE_ABSTRAIT est positionné, ou plus généralement, le champ SELECTIONNABLE est positionné.

Réalisation abstraite

Idée intuive

L'idée consiste à séparer l'algorithme en deux parties. La première recherche dans la S.I.G. la boîte possesseur du point qui est la moins composante c'est-à-dire dont le niveau est le plus haut dans la hiérarchie. La deuxième partie se charge de vérifier la définition de la granularité à partir de la boîte possesseur du point la moins composante.

DETERMINER_BOITE_DESIGNEE_POINT (SIG_désignée,point_désigné)

début
boîte_poss:= **RECHERCHE_BOITE_POSSESSEUR** (SIG_désignée,point_désigné)
boîte_désignée := **VERIFIEE_GRANULARITE** (boîte_poss,point_désigné)
retourner (boîte_désignée)
fin

avec

RECHERCHE_BOITE_POSSESSEUR (boîte, point)

début
boîte_composante = **RECHERCHE_DANS_BOITES_SUIVANTES** (boîte, point)
Si (boîte_composante = VIDE)
Alors
retourner (boîte)
Sinon
retourner (**RECHERCHE_BOITE_POSSESSEUR** (boîte_composante, point))
finsi
fin

et

RECHERCHE_DANS_BOITES_SUIVANTES (boîte, point)

Soit L la liste des boîtes suivantes de boîte.

```
début
    Pour tout  boîte de L faire
        Si     le point appartient à la boîte
               Alors
                   retourner (boîte)
        fsi
    ffaire
    retourner (VIDE)
fin
```

3) Recherche en fonction de la granularité

Spécification

Cette fonction a pour effet de déterminer dans l'ensemble des boîtes possesseur du point, la boîte désignée dont la granularité est la plus fine.

- *arguments :*
  boîte_poss : identificateur de la boîte la moins composante parmi les boîtes possesseur du point désigné
  point_désigné : position d'un point à l'écran

- *pré-condition :*
  boîte_poss identifie une boîte existante

- *résultat :*
  boîte_désignée : identificateur de la boîte désignée

- *post-conditions :*
  Si l'identificateur boîte_désignée est VIDE,
  alors il n'existe pas de boîte désignée par la position écran point_désigné,
  sinon boîte_désignée identifie la boîte désignée définie comme suit : c'est une boîte soit "sélectionnable", soit boîte abstraction. Pour les boîtes textuelles, cette boîte est possesseur du point désigné, et pour les boîtes graphiques, le point désigné est inclus dans les limites du concept graphique représenté. Cette boîte est la moins composante des boîtes désignées, son niveau dans la hiérarchie est inférieur ou égal à celui de la boîte identifiée par boîte_poss.

Réalisation abstraite

Idée intuive

L'idée consiste à remonter dans la hiérarchie à partir de la boîte identifiée par boîte_poss pour déterminer la boîte désignée la moins composante. En effet, boîte_poss est la boîte possesseur du point la moins composante.

```
RECHERCHE_GRANULARITE (boîte_poss, point_désigné)

début
        boîte_retournée := boîte_poss;
        convient := VRAI;

        Si (boîte_poss n'est pas VIDE)
        Alors
                Si (boîte_poss n'est pas une boîte abstraction)
                Alors
                        Si      (boîte_poss possède un lien avec arbre abstrait) ou
                                (boîte_poss est sélectionnable)
                        Alors
                                Si      (boîte_poss est une boîte graphique)
                                Alors
                                        Si      (le point_désigné œ aux limites du concept graphique)
                                        Alors
                                                boîte_retournée : = VIDE
                                        finsi
                                Sinon
                                        Si (boîte_poss est une inter-boîte)
                                        Alors
                                                boîte_retournée : = VIDE
                                        finsi
                                finsi
                        Sinon
                                convient = FAUX
                        finsi
                finsi
        Sinon
                boîte_retournée : = VIDE
        finsi

        Si      convient = VRAI
        Alors
                retourner (boîte_retournée)
        Sinon
                retourner (RECHERCHER_GRANULARITE (précédent (boîte_retournée),
                point_désigné))
        finsi
fin
```

6.11.4.4 Algorithme d'annotation

La méthodologie que nous proposons, pour la réalisation de l'annotation s'est inspirée de différentes méthodes existantes. Bien qu'aucune de ces méthodes ne corresponde vraiment à

notre problème, elles mettent toutefois en évidence une division en différentes étapes, qui sont le calcul de coordonnées relatives et le calcul de coordonnées absolues.
Nous présentons en premier lieu ces méthodes, ensuite notre proposition de solution.

1) Méthode des degrés relatifs : principe heuristique

Cette **heuristique** proposée dans [Delarche 79], pour la production automatique de graphes hiérarchisés est résumée dans [Lefèvre 87]. Nous en donnons une brève description.
Cette heuristique a pour but de minimiser le nombre de croisements d'arcs dans les graphes hiérarchisés. Elle a d'abord été proposée pour les graphes bipartis (deux niveaux) et ensuite généralisée pour les graphes comportant plusieurs niveaux.
Pour essayer de ne pas avoir trop de croisements d'arcs sur le dessin, une idée naturelle est de chercher à rendre les arcs les plus verticaux possibles. C'est-à-dire étant donné une permutation P1 des sommets du premier niveau, à construire une permutation P2 qui place les sommets du bas le plus possible en-dessous des sommets du point auxquels ils sont reliés.
Pour cela, nous attribuons des abscisses aux sommets du premier niveau et nous calculons des abscisses pour les sommets du deuxième niveau. Chaque abscisse a une valeur comprise entre 0 et 1, ce qui permet de travailler indépendamment de l'espace physique dont nous disposons pour dessiner le graphique. Le calcul des abscisses du premier niveau consiste à faire la moyenne des abscisses des sommets du premier niveau auxquels ils sont reliés.
La généralisation de cette méthode à plusieurs niveaux pose un certain nombre d'inconvénients :

- la méthode se borne à limiter le nombre de croisements situés localement entre deux niveaux successifs, sans tenir compte des répercussions de ce choix sur l'ensemble de la représentation. Il n'y a pas d'optimisation globale dans la recherche du nombre minimum de croisements,
- cette méthode peut produire deux valeurs d'abscisses identiques pour deux sommets d'un même niveau,
- cette méthode accepte difficilement des contraintes de disposition supplémentaires comme par exemple imposer qu'un sommet se trouve à côté ou au-dessus d'un autre. Ce qui la rend inapplicable à notre problème étant donné ces contraintes qui sont exprimées dans les inter-boîtes.

2) Méthode proposée par [Rowe 87]

La méthode proposée ici est une amélioration de l'algorithme de [Sugiyama 81] pour la représentation d'un graphe en vue de permettre des cycles et d'améliorer la représentation. Cet algorithme fonctionne en trois phases.
La **première phase** consiste à convertir le graphe de départ en une **hiérarchie** dans laquelle les arcs ne se croisent pas de plus d'un niveau (cfr. la construction d'une hiérarchie propre dans [Lefèvre 87]) . Cette première phase assigne un numéro de niveau à chaque sommet du graphe.
Celle-ci est réalisée en plusieurs étapes.
En premier lieu, il s'agit d'assigner des niveaux de telle sorte que chaque sommet soit à un niveau inférieur à ses ancêtres. Il s'agit de réaliser la fermeture transitive de tous les descendants d'un sommet pour déterminer quels sont les sommets qui n'ont pas de descendant

et qui appartiennent donc au sommet du graphe. A ces sommets nous assignons le niveau courant et nous le retirons de la fermeture ainsi que ses arcs. Nous répétons cette opération jusqu'à ce que tous sommets aient un niveau.
La seconde étape de cette première phase consiste à "casser" les arcs qui traversent plus d'un niveau en plaçant des sommets supplémentaires aux niveaux intermédiaires.
Les cycles sont cassés en inversant temporairement l'arc qui termine le cycle.

La **deuxième phase** de l'algorithme trie les noeuds à chaque niveau pour minimiser le nombre de croisements. Cette deuxième phase se résume à exprimer les positions relatives d'un sommet suivant ses barycentres. Le **up-barycentre** (**down-barycentre**) d'un sommet est la position moyenne de ses précédents (suivants) incluant les sommets supplémentaires introduits.
Cette deuxième phase fonctionne en plusieurs passes.
La première passe trie les sommets suivant leur up-barycentre en partant de la racine vers le bas.
La seconde passe trie les sommets suivant leur down-barycentre en partant du bas vers la racine.
Cette deuxième phase se termine soit lorsqu'il n'y a plus de croisements d'arcs, soit après un certain nombre de passes.

La **troisième phase** consiste à donner les positions absolues des sommets c'est à dire les coordonnées x et y du sommet. Cette troisième phase se fait également en plusieurs passes pour tenir compte de l'espacement entre les sommets d'un même niveau et de niveaux différents.
Cette méthode suppose en entrée un graphe dont les arcs représentent les liens entre les différents sommets. Ces liens étant directement représentables, il existe une correspondance 1-1 entre le graphe et ce qui est affiché à l'écran.
Pour réaliser la deuxième phase, il est nécessaire parfois de changer l'ordre des sommets sur un même niveau. Ceci afin de minimiser le nombre de croisements. Dans notre cas, ce changement n'est pas acceptable dans la mesure où l'ordre est donné par l'utilisateur. En ce sens la méthode ne tient pas compte des orientations NORD, etc. de notre S.I.G..

3) Proposition de solution

La difficulté de trouver une bonne méthode pour réaliser cette annotation repose sur le fait que nous n'avons pas une relation 1-1 entre le graphe qui est représenté à l'écran et la S.I.G.. En effet, nous n'avons pas réussi à exprimer notre S.I.G. sous forme d'un graphe, dont les arcs représenteraient les liens (d'orientation) entre les boîtes qui sont sommets du graphe. Ces arcs tiendraient compte de l'orientation de l'inter-boîte qu'ils représentent.
Néanmoins, ces différentes méthodes mettent en évidence les étapes de positionnement relatif et de positionnement absolu dans la réalisation de l'annotation. Notre proposition de solution suppose également plusieurs étapes. Chacune d'entre elles consiste à ajouter une information supplémentaire sur la S.I.G..
La première d'entre elles se charge du calcul des unités.
La seconde consiste à attribuer pour chaque boîte de la S.I.G., des coordonnées relatives à l'espace d'affichage disponible. Ces coordonnées relatives s'expriment sous forme d'une fraction de l'espace d'affichage comprise entre 0 et 1.

La dernière attribue des coordonnées absolues en fonction des coordonnées relatives précédemment obtenues, et en fonction de critères d'espacement entre les boîtes.

Nous présentons maintenant les différentes étapes qui n'ont pas pu être approfondies. Nous en donnons les principes et expliquons brièvement un des problèmes rencontrés qui est celui des **croisements** d'objets graphiques.

Le calcul des unités

Nous avons vu, dans la section 6.11 que ce problème se résume à trouver un "chemin" entre deux boîtes. Un tel chemin ne se retrouve pas tel quel dans la S.I.G.. Il faut, en effet, le construire à partir des inter-boîtes. Une idée consiste à construire à partir de la S.I.G., un graphe dont les sommets sont des boîtes et les arcs sont des inter-boîtes. Dans ce graphe, seules les notions d'extrémité initiale et terminale de l'inter-boîte peuvent être représentées. Comme déjà dit, il n'a pas été possible, en effet, de représenter à l'aide d'un arc la notion d'orientation de l'inter-boîte (cfr. le champ attribut-composition). Nous pouvons alors construire la **matrice d'accessibilité** du graphe et à partir de celle-ci, réaliser facilement le concept d'unité. Pour savoir à quel sommet du graphe correspond une boîte de la S.I.G. et vice versa, il est nécessaire de faire la correspondance entre la S.I.G. et le graphe.
La matrice d'accessibilité indique pour chacun des sommets du graphe, s'il est accessible d'un autre ainsi que le nombre de chemins joignant les deux sommets. Etant donné une boîte de la S.I.G., pour savoir s'il existe un chemin joignant cette boîte à une autre, il suffit de voir à quels sommets elles correspondent et de voir dans la matrice d'accessibilité s'il existe un chemin les joignant.

Le calcul des coordonnées relatives

Objectif

L'objectif premier de cette étape est d'être indépendant de l'espace d'affichage disponible. En effet, ces coordonnées s'expriment sous forme d'une fraction de cet espace quelque soit celui-ci.
En second lieu, l'attribution de ces coordonnées permet de réaliser les différents critères de centrage vus auparavant.

Principe : première proposition

Ces coordonnées s'expriment sous forme de fractions de l'espace disponible; une fraction de la largeur et une fraction de la hauteur disponible. Ces fractions doivent respecter les critères de centrage . Les coordonnées relatives d'une boîte, seront donc fonction des coordonnées relatives des boîtes auxquelles elle est reliée.
L'attribution de ces coordonnées relatives (abscisse, ordonnée relatives) repose sur la notion de division de l'espace d'affichage. Le principe en est le suivant :

- de par ses coordonnées relatives, chaque boîte détermine l'espace disponible pour les boîtes qui lui sont rattachées. Ces boîtes se répartissent de manière égale cet espace. Ce qui assure par la même, le critère de centrage des boîtes,

- de par ses coordonnées relatives, chaque boîte détermine également l'espace d'affichage disponible qui lui est associé pour sa représentation.

Pour attribuer ces coordonnées, nous devons parcourir la S.I.G. dans un certain ordre. Ce parcours est basé sur le concept de boîte mère-unité.
Nous cherchons en premier lieu les coordonnées relatives de la boîte mère-unité. Ensuite, nous cherchons les coordonnées relatives des boîtes filles-unité. Ces boîtes doivent se répartir également l'espace déterminé par les coordonnées de la boîte mère-unité. Par exemple, dans la cas où ces coordonnées sont (1/2, 1/2) cela signifie que les boîtes filles-unité disposent d'un espace d'affichage qui a comme hauteur 1 et comme largeur 1. Plus précisément, les boîtes situées au SUD ou au NORD, disposent d'un espace de hauteur 1 et de largeur 1/2, et les boîtes situées à l'EST ou à l'OUEST, disposent d'un espace de hauteur 1 et de largeur 1/2.

boîte-mère-unité
abscisse relative : numérateur = 1 et dénominateur = 2
ordonnée relative : numérateur = 1 et dénominateur = 2

Supposons que la boîte mère-unité dispose de deux boîtes filles-unité qui soient situées au SUD. Le fait d'ajouter ces boîtes filles-unité a plusieurs conséquences :

- ces boîtes doivent être centrées par rapport à la boîte mère-unité. Nous devons faire une nouvelle division de la largeur de l'espace disponible (c'est-à-dire 1) entre les boîtes filles. L'abscisse relative s'exprime maintenant en quart d'espace (dénominateur = 4),

- cette modification doit être répercutée sur la boîte mère-unité. En effet, l'ajout de boîtes situées au SUD a augmenté le nombre de niveaux que doit contenir l'espace d'affichage. Ce nombre est monté à deux. La hauteur de l'espace doit donc être répartie en deux niveaux. Ce qui donne :

boîte mère-unité
ordonnée relative : numérateur = 2 et dénominateur = 4
boîte fille-unité
ordonnée relative : numérateur = 3 et dénominateur = 4

En fait, l'ajout de boîtes ne change pas la valeur de la fraction exprimant la coordonnée relative elle change seulement le numérateur et le dénominateur. Ce qui permet d'ajuster par la même l'espace disponible pour cette boîte et pour les boîtes rattachées.

Cette première solution a comme avantage de se réaliser assez facilement. Elle est basée sur une division de l'espace d'affichage donnant pour chaque boîte son espace disponible. Elle est

intéressante pour une annotation incrémentale puisque les coordonnées d'une boîte ne dépendent que de la boîte mère-unité.
Cependant, cette proposition de solution privilégie les premières boîtes qui sont considérées par le calcul. Plus précisemment, elle accorde plus de place pour les boîtes mères-unité que pour les boîtes filles-unité, et par conséquent elle ne permet pas une bonne occupation de la place d'affichage disponible. Cet inconvénient pourrait être soulevé dans une deuxième proposition de solution présentée brièvement ci-après.

Principe : deuxième proposition

Nous n'approfondissons pas beaucoup plus cette proposition. Seuls quelques principes seront exposés. Cette proposition fonctionne en plusieurs passes. La première passe consiste à déterminer la hauteur et la largeur maximales de la S.I.G..
En même temps, au cours de cette passe chaque boîte se voit attribuer deux entiers supérieurs à 0 correspondant à son niveau_Y (position en Y) et à son niveau_X (position en X) dans un plan cartésien. La deuxième passe détermine les ordonnées et abscisses relatives de la boîte à partir de son niveau_Y et de son niveau_X dans le plan et à partir de la hauteur et largeur maximales de la S.I.G.. En effet, la hauteur (largeur) maximale définit le dénominateur commun des ordonnées (abscisses) relatives. Le niveau_Y (niveau_X) de la boîte définit le numérateur de l'ordonnée (abscisse) relative.

3) Le calcul des coordonnées absolues

Principe

Ayant attribué les coordonnées relatives, nous devons maintenant poser des coordonnées absolues. Le principe est le suivant :

- les coordonnées relatives d'une boîte, une fois transformées en coordonnées absolues déterminent la position d'un point écran par rapport auquel la boîte doit être centrée. Elles déterminent également l'espace disponible pour la représentation de cette boîte.

Pour connaître la position des points supérieur droit et inférieur gauche de la boîte, nous devons en premier lieu connaître les dimensions hauteur et largeur de celle-ci. Ensuite, compte tenu de l'espace disponible, nous devons voir si cette boîte peut être représentée à l'écran.
Pour une boîte élémentaire, sa taille peut facilement être connue de même que pour une boîte textuelle car il s'agit de sommer la taille des boîtes composantes. Pour une boîte graphique, cette taille va également dépendre des boîtes composantes. Cependant, lorsqu'une inter-boîte est composante de cette boîte graphique, la taille de celle-ci ne peut être directement connue. Il faut d'abord poser les coordonnées absolues des boîtes composantes, ensuite poser les coordonnées des inter-boîtes, et enfin calculer la taille de la boîte graphique.
Ce principe, repose sur un certain parcours de la S.I.G.. Dans ce parcours, plusieurs cas sont envisageables suivant que la boîte est représentable ou qu'il faille envisager la boîte abstraction.
Etant donné que les coordonnées relatives découpent l'espace d'affichage en sous-espaces, il se peut qu'il y ait des problèmes d'espacement entre les boîtes. En effet, deux boîtes, l'une à côté de l'autre, peuvent être représentables chacune dans son sous-espace d'affichage mais peuvent

ne pas respecter l'espacement minimal entre les boîtes. Pour résoudre ce problème, il suffit de considérer les sous-espaces d'affichage plus petits qu'ils ne le sont réellement. Autrement dit, il suffit de les considérer diminués en largeur et en hauteur, de la moitié de cet espacement minimal. Comme les deux sous-espace sont limitrophes, ils réalisent bien ensemble l'espacement minimal (1/2 + 1/2 = 1).

Parcours de la S.I.G.

L'idée du parcours de la S.I.G. est basée sur le concept d'unité. Il s'agit de parcourir en premier la boîte mère-unité et ensuite les boîtes filles-unité correspondantes. A chaque boîte rencontrée, nous nous demandons si elle est représentable dans son sous-espace disponible, autrement dit nous nous demandons si le sous-espace est suffisant que pour dessiner la boîte et ses composantes. Une boîte représentable ne signifie pas pour autant qu'elle sera dessinée à l'écran car cela dépend de ses boîtes filles-unité et des inter-boîtes qui les relient.
Si la boîte mère-unité n'est pas représentable, il est inutile d'essayer de représenter les boîtes filles-unité de cette boîte. Auquel cas celle-ci peut être représentée par une boîte abstraction. Ce qui a comme conséquence que nous devons complèter la représentation de l'unité à laquelle la première boîte appartient.
Si par contre elle est représentable, il faut encore voir si ses boîtes filles-unité sont représentables et auquel cas si lorsque nous posons les inter-boîtes l'unité est encore représentable.

4) Le problème des croisements

Lors de l'attribution des coordonnées absolues, des croisements de boîtes peuvent se produire. Nous parlons de croisements lorsque des graphiques (boîtes et/ou inter-boîtes) se superposent. Ces croisements sont dûs non seulement au fait que nous avons un graphe à l'écran mais également au fait que les boîtes peuvent se placer au SUD, à l'EST etc..
Pour résoudre le problème des croisements, nous devons en premier lieu les détecter et ensuite essayer de les minimiser. Il est en effet, impossible de les éliminer puisque nous représentons dans la majorité des cas un graphe non planaire.
La détection de ces croisements est un problème assez conséquent et ne peut être résolu dans le cadre de ce mémoire.
Pour les diminuer, une idée consiste à utiliser la boîte abstraction pour remplacer les boîtes qui font l'objet de ces croisements.

6.11.5 Conclusion

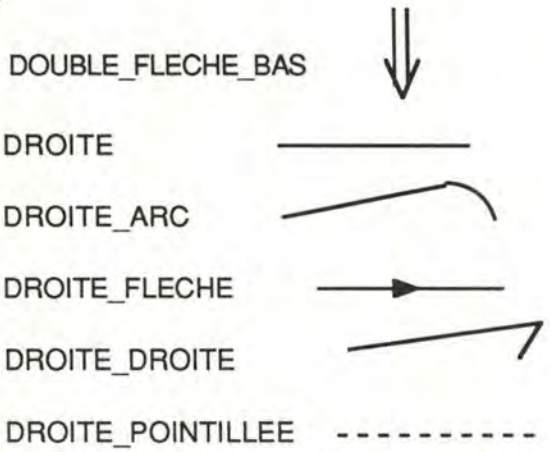
Nous avons proposé une suite d'étapes qui permet de réaliser l'annotation de la S.I.G.. Chacune de ces étapes doit faire l'objet d'une étude plus détaillée pour s'assurer qu'elles respectent les conventions de représentation graphique ainsi que les critères esthétiques que nous nous sommes fixés.

6.12 Module d'affichage de la structure intermédiaire

Ce module sera chargé d'afficher la structure intermédiaire avec notion de positions dans une fenêtre. Le module devra donc uniquement avec les informations contenues dans la structure intermédiaire, afficher celle-ci dans une fenêtre. La structure intermédiaire contient un certain nombre de références à des concepts graphiques pour lesquels le module d'affichage devra consulter la table des concepts graphiques pour connaître le nom de la procédure d'affichage à appeller pour afficher le concept graphique en question. Avant d'afficher un noeud de la structure intermédiaire graphique à l'écran, le module d'affichage devra vérifier que le noeud n'a pas été annoté comme impossible à représenter pour cause de place par le module d'annotation de la structure intermédiaire graphique.

Les concepts graphiques minimum nécessaires pour représenter l'arbre des types d'un type d'une spécification SACSO sont les suivants :

- pour les objets : RECTANGLE
- pour les relations :



On pourrait aussi imaginer un certain nombre de concepts de base qu'il serait possible de composer pour obtenir de multiples représentations possibles.

Spécification pré-post du module d'affichage

affichage (FENETRE, STRUCT_INT_POS, TDCG)

- arguments :
  FENETRE : fenêtre dans laquelle doit être affichée la structure intermédiaire
  STRUCT_INT_POS : structure intermédiaire avec notion de positions représentant l'information nécessaire pour l'affichage de la représentation graphique
  TDCG : table des concepts graphiques

- pré-conditions :
- résultats :
- post-conditions :

La structure intermédiaire STRUCT_INT sera affichée dans la fenêtre FENETRE. Tous les noeuds de la structure intermédiaire seront affichés à la position qui leur a été attribuée par le module d'annotation de la structure intermédiaire.

6.13 Exemples de structures logiques

Nous donnons dans cette section, quelques exemples de structures logiques pouvant être utilisées pour la définition d'une relation dans la section DEFINITIONS_RELATIONS du langage de description des objets et des relations. En fait ces structures logiques utilisent comme dit précédemment les constructeurs de types SACSO. Les opérations possibles sur les constructeurs que nous présentons sont un sous-ensemble des opérations SACSO et concernent uniquement la consultation. Toutefois dans une optique de généralité, il est possible d'adjoindre toutes les opérations SACSO possibles sur les constructeurs de types.
Les noeuds du type abstrait "ARBRE ABSTRAIT" peuvent être structurés de différentes manières :

1) liste ou suite de noeuds de même type
exemple : LIST_NOEUD = S[BLOC-REL]
Les opérations utiles sont les suivantes :
- app : S[**F-ELEM**], **F-ELEM -> BOOL**
  appartenance d'un élément donné à une suite
- taille : S[**F-ELEM**] -> **ENTIER**
  taille, nombre d'éléments de la suite
- acc : S[**F-ELEM**], **ENTIER -> F-ELEM**
  accès dans une suite à un élément donné par son rang
- tête : S[**F-ELEM**] -> **F-ELEM**
  accès au premier élément de la suite

2) table de noeuds
exemple : TABLE_NOEUD = T[nom, BLOC-REL]
Les opérations utiles sont les suivantes :
- appt : T[**F-ELEM, F-ELEM1**], **F-ELEM -> BOOL**
  appartenance d'un indice donné dans une table
- taillet : T[**F-ELEM, F-ELEM1**] -> **ENTIER**
  taille (nombre d'indices accessibles) d'une table
- acct : T[**F-ELEM, F-ELEM1**], **F-ELEM -> F-ELEM1**
  accès dans une table à un élément donné par son indice

3) produit cartésien de noeuds
exemple : PC_NOEUD = PC [BLOC-REL, BLOC-OP]
Les opérations utiles sont les suivantes :
- accès : PC[**F-ELEM, +**], **ENTIER -> F-ELEM1**
  accès à la valeur d'un champ d'un produit cartésien

4) union de noeuds
exemple : U_NOEUD = U[BLOC-OP]
Les opérations utiles sont les suivantes :
- defaire : U[**F-ELEM, +**], **ENTIER -> F-ELEM1**
  donne une partie d'une union
- est-type : U[**F-ELEM, +**], **F-ELEM1 -> BOOL**
  permet de savoir de quel type est une union

6.14 Algorithmes abstraits des différents modules

Cette section présente les algorithmes abstraits des modules les plus importants qui ont été spécifiés par pré-post précédemment.

6.14.1 Module de saisie

saisie (TDOR, type_objet, relation, noeud_arbre_abstrait)

```
Début
        événement = lire-événement-dans-file ();
        si événement = clic-souris-milieu
        alors
                événement = lire-événement-dans-file ();
                si événement = représentation-graphique
                alors
                        obtenir-position-souris (x, y);
                        si existe-dans-sit (objet, x, y)
                        alors
                                noeud_arbre_abstrait = noeud_AA_référencé_par
                                                        (objet);
                                type-objet = déterminer-type (objet);
                                si ((no-ligne =
                                  appartient (TDOR, type-objet, "nom-objet") <> 0)
                                alors
```

```
                    relations =
                        consult (tdor, no-ligne, "nom-objet");
                    afficher (relations);
                    relation = obtenir-sélection-utilisateur ();
                sinon
                    afficher-erreur ("Aucune relation n'a été définie
                        pour le type d'objet 'type_objet')
    Fin
```

6.14.2 Module de recherche de l'information dans l'arbre abstrait et de construction de la structure intermédiaire graphique

recherche_construction (arbre-abstrait, noeud-départ, noeud-racine, table, struct-int-graph)

Début

```
    boîte-racine := créer-boîte-racine (struct-int-graph);
        {créer le noeud racine de la structure intermédiaire}
    list-noeud := creer-liste (noeud-départ);
    créer-reste-sig (struct-int-graph, boîte-racine, list-noeud, table, arbre-abstrait, noeud-
                racine);
```

Fin

créer-boîte-racine (struct-int)

Début

```
    boîte-racine := créer-noeud-racine ();
    struct-int := boîte-racine;
    remplir-noeud (boîte-racine, noeud-départ, struct-int, table);
        {remplir tous les champs du noeud boîte-racine}
```

Fin

créer-reste-sig (sruct-int, noeud-père, list-noeud-AA, table, arbre-abstrait, noeud-racine)

```
Début
        si list-noeud-AA vide
        alors
        sinon
                tant que list-noeud-AA <> vide
                faire
                        noeud-fils := construire-noeud-fils (struct-int, noeud-père);
                                {construire dans la structure intermédiaire struct-int un noeud
                                 fils du noeud noeud-père}

                        remplir-noeud (noeud-fils, premier (list-noeud-AA), struct-int, table);
                                {remplir les champs du noeud noeud-fils}

                        nouvelle-liste := appliquer-relation (table, premier (list-noeud-AA),
                                        arbre-abs, noeud-rac);
                                {nouvelle-liste est obtenue en appliquant la définition de la
                                 relation se trouvant dans la table table au premier noeud de la
                                 liste list-noeud-AA}

                        construire-inter-boîte (struct-int, noeud-père, noeud-fils,
                                                premier (nouvelle-liste), table);
                                {les noeuds inter-boîte éventuels à construire sont ajoutés dans
                                 la structure intermédiaire et remplis}

                        liste-noeud-AA := liste-noeud-AA - premier (liste-noeud-AA);
                        créer-reste-sig (struct-int, noeud-père, nouvelle-liste, table,
                                        arbre-abs, noeud-rac);
                fin faire
Fin
```

CONCLUSION

Conclusion

L'approche adoptée dans ce mémoire a été de concevoir des outils graphiques qui répondent à des besoins plus généraux que les besoins spécifiques de SACSO. En effet, ces outils peuvent être réutilisés en dehors du cadre de SACSO. Ainsi, la petite boîte à outils que nous avons construite profite du statut de standard de fait de X Windows. Elle peut dès lors être utilisée sur un grand nombre de systèmes.
L'outil de visualisation dirigé par des tables peut :

- être adapté à une structure physique d'arbre abstrait particulière en modifiant une table (la table des correspondances entre structures logiques et physiques),

- se voir adjoindre de nouveaux concepts graphiques en ajoutant leurs descriptions dans une autre table (la table des concepts graphiques),

- être paramétré sur la définition d'un langage d'interface graphique (L.I.G.) car un méta-langage, le L.D.O.R. combiné au langage graphique permet de définir un L.I.G. effectif dont la description est placée dans une table (la table de description des objets et des relations).

1. Les apports

Suite au portage du système SACSO sur une station de travail, nous avons réalisé un nouveau multi-fenêtrage. Ce dernier profite pleinement des possibilités graphiques plus étendues des stations de travail. Ce multi-fenêtrage a été construit grâce à une petite boîte à outils. Cette boîte à outils que nous avons construite avec X Windows est réutilisable sur de nombreux systèmes car elle profite du statut de standard de fait de X Windows.

L'originalité de ce mémoire est d'avoir proposé un langage (le L.D.O.R.) permettant de préciser les objets d'un arbre abstrait et les relations entre ces objets dont une représentation graphique est souhaitée à l'écran. Ce langage permet de définir un langage d'interface graphique (L.I.G.) dans la mesure où il est possible de préciser à la fois les aspects sémantiques, lexicaux et syntaxiques d'un L.I.G..

D'autre part, un outil de visualisation se voulant général, ne peut se limiter à des concepts graphiques prédéfinis. Dans notre outil, pour cette raison, il est possible d'adjoindre la description de nouveaux concepts graphiques.

Le deuxième apport de ce mémoire est d'avoir proposé des solutions aux problèmes complexes d'affichage d'une représentation graphique dans un espace de taille limitée. De plus, nous avons proposé un squelette de structure intermédiaire permettant de modéliser un grand nombre de représentations graphiques de par ses abstractions de boîte et d'inter-boîte.

Signalons finalement que malgré la multitude d'éditeurs graphiques existants, on trouve très peu d'outils de visualisation graphiques à vocation générale dans la littérature. La plupart des outils existants, sont destinés à un usage spécifique. Ceci illustre par la même, la difficulté de construire un outil de visualisation graphique à vocation générale.

2. Les limites

L'outil de visualisation n'a pas encore fait ses preuves. Il reste à évaluer le langage de description des objets et des relations. Il convient de s'assurer de sa puissance d'expression pour décrire des représentations graphiques d'objets et de relations modélisés dans un arbre abstrait. D'autre part, plusieurs problèmes épineux tels l'annotation incrémentale et le réaffichage incrémental restent à approfondir.

Seule une étude ultérieure approfondie et des expériences de prototypage et d'implémentation pourront permettre de juger du caractère réaliste et faisable, sur le plan des performances, de certaines propositions faites ici. Ceci permettra en outre de déterminer des meilleurs compromis entre généralité et efficacité.

3. Les extensions possibles

La boîte à outils, telle que conçue actuellement, devrait être étendue pour permettre la gestion de concepts graphiques (rectangle, cercle ...). Cette extension de la boîte à outils permettrait son utilisation par l'outil de visualisation.

Ce dernier, une fois les limites énoncées précédemment comblées, pourrait être transformé en éditeur graphique dirigé par la syntaxe. Nous avons esquissé les modifications qu'il faudrait apporter au langage de description des objets et des relations. Cette transformation de l'outil de visualisation en éditeur graphique permettrait :

- la prise en compte des deux sens de l'interaction graphique, à savoir la sortie graphique et l'entrée graphique,

- la modification par l'utilisateur de la représentation graphique lorsque l'algorithme de positionnement (lay-out) automatique est pris en défaut.

De plus, l'éditeur profiterait des possibilités de positionnement (lay-out) automatique de l'outil de visualisation.

Les résultats obtenus concernant l'outil de visualisation ne sont sans doute pas à la hauteur des résultats espérés. Ils ont toutefois le mérite d'avoir apporté des solutions à quelques problèmes épineux.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[Brossard 86]
Brossard M-J, Conchon A, Morcos E (1986) *Affichage interactif d'arbres abstraits.*
Présentation technique, 4-5-6 Février 1986. Palais des congrès, Perros-Guirec

[Chailloux 86]
Chailloux J (1986) *Le_Lisp version 15.2, le manuel de référence.*
INRIA, Rocquencourt

[Colnet 86a]
Colnet D (1986) *De CEYX.*
Centre de Recherches en Informatique de Nancy, 1986

[Colnet 86b]
Colnet D, Masini G, Napoli A, Noiret Y, Tombre K (1986) *Les langages orientés objets.*
Centre de Recherches en Informatique de Nancy, rapport interne, 1986

[Coutaz 85]
Coutaz J (1985) *Abstractions pour Interfaces Interactives.*
TSI, volume 5, n°4, 1986

[Dubois 82]
Dubois E, Finance J-P, Van Lamsweerde A (1982) *Towards a Deductive Approach to Information System Specification and Design.*
International Symposium on Current Issues of Requirements Engineering Environments

[Dubois 84]
Dubois E (1984) *Cadre et méthodes de spécification de systèmes d'informations fondés sur les types abstraits.*
Thèse de Docteur-Ingénieur INPL Nancy

[Dubois 85]
Dubois E, Finance J-P, Van Lamsweerde A (1985) *A Constructive Approach to Requirements Specification.*
Manuscript 101, Phillips Research Laboratory Brussels.

[Dubois 86]
Dubois E, Lévy N, Souquières J (1986) *SACSO : Méthodes et Outils de Construction de Spécifications de Systèmes.*
Actes CGL3 - Versailles, mai 1986

[Dubois 87a]
Dubois E, Lévy N, Souquières J (1987) *Définition de Transformations de Structures dans le Processus de Construction d'une Spécification.*
ESEC 1987, Strasbourg

[Dubois 87b]
Dubois E, Van Lamsweerde A (1987) *Making Specification Processes Expilicit.*
Proccedings 4th International Workshop on Software Specification and Design IEEE, Monterey, april 3-4, 1987

[Gettys 86]
Gettys J, Scheifler R (1986) *The X Window System.*
ACM Transactions on Graphics, volume 5, n°2, April 1986, pages 79-109

[Gruia 85]
R. Gruia-Catalin *A Taxonomy of current Issues in Requirements Engeneering.*
IEEE Computer april 1985.

[Hullot 84]
Hullot J-M (1984) *CEYX version 4, le manuel de référence.*
INRIA, Rocquencourt

[Kernighan 86]
Kernighan B, Ritchie D (1986) *Le langage C.*
Editions Masson, 1986

[Kernighan 86]
Kernighan B, Pike R (1986) *L'environnement de programmation UNIX.*
Editions InterEditions, 1986

[Klint 83]
Klint P (1983) *A Survey of three Language-Independent Programming Environments.*
Stichting Mathematisch Centrum, Amsterdam

[Lévy 87]
Lévy N, Piganiol A, Souquières J (1987) *Specifying with SACSO.*
Proccedings 4th International Workshop on Software Specification and Design IEEE, Monterey, april 3-4, 1987

[Masini 84]
Masini G (1984) *Tout ce que vous avez toujours voulu savoir sur Lisp et que vous osez enfin demander.*
Centre de Recherches en Informatique de Nancy, 1984

[Mélèse 82]
Mélèse B (1982) *METAL, un langage de spécification pour le système MENTOR.*
Technique et Science Informatique (T.S.I.), volume 1, n°4, 1982

[Nanard 84]
Nanard J, Nanard M (1984) *Manipulation interactive de documents*
Technique et Science Informatique (T.S.I.), volume 3, n°6, pages 443-451,1984

[Nerson 85]
Nerson J-M, Meyer B, Soon Hae Ko (1985) *Showing programs on a screen.*
Science of Computer Programming, volume 5, n°2, pages 111-142, 1985

[Lefèvre 87]
Lefèvre J, Sacré B (1987)*Interprétation graphique des mesures de performances d'un système.*
Mémoire, FUNDP, Institut d'informatique, 1987

[Rowe 87]
L.A. Rowe, M. Davis, E. Messinger, C. Meter, C. Spirakis, A. Tuan *A Browser for Directed Graphs.*
Software-Practice and experience, vol 17(1), 61-67 (january 1987).

[Schneiderman 87]
Schneiderman B (1987) *Designing the User Interface. Strategies for effective human-computer interaction.*
Addison-Wesley, 1987.

[Souquières 86]
Souquières J, Valdenaire A (1986) *Environnement de SACSO.*
Revue Génie Logiciel n°6, novembre 1986

[Sugiyama 81]
K. Sugiyama, S. Tagawa, M Toda *Methods for visual understanding of hierarchical system structures.*
IEEE Transaction on Systems Man, and Cybernetic, SMC-11, 109)125 (1981).

[Sun 87]
Sunview Programmer's Guide.
Sun Microsystems, Mountain View, CA 94043. Revision A of 17 February 1986

[Van Lamsweerde 85]
Van Lamsweerde (1985) *Cadre général pour un modèle de cycle de vie d'un projet informatique*
FUNDP, Namur, Institut d'informatique, rapport interne, 1985

[Wasserman 79]
Wasserman A, Stinson S (1979) *A Specification Method for Interactive Information Systems*
Proceedings I.E.E.E. Computer Society Conference, Specification of reliable Software, 1979

[Wasserman 85]
Wasserman A (1985) *Extending State Transition Diagrams for the Specification of Human-Computer Interaction*
IEEE Transactions on Software Engineering, volume n°8, August 1985

[XWindows 87]
Programmers Guide, version 10.

ANNEXES

Annexe 1

Cette annexe reprend les spécifications par pré-post de toutes les primitives de la boîte à outils construite en C - X Windows. Les primitives sont regroupées par fichier contenant leur code source. A l'intérieur de chaque fichier les primitives sont regroupées par objet X Windows (fenêtre, éditeur à boutons, menu ...)

Les procédures sont distinguées par un (P) et les fonctions par un (F). Pour chaque primitive est indiquée en note les primitives X Windows qu'elle utilise.

A1.1 Fichier PRIMX.C

1.1.1 Primitives d'initialisation de l'écran et de la librairie Xrlib contenant les primitives X Windows pour les menus, boîtes à messages ...

+ OUVRIR_ECRAN (F)

- argument :

- pré-condition :
  OUVRIR_ECRAN n'a pas encore été exécutée

- résultat :
  ECRAN : pointeur vers l'écran qui a été ouvert

- post-conditions :
  L'écran par défaut c'est-à-dire NULL (écran de la machine console) a été ouvert. Une connexion avec le serveur X est établie. En effet, le serveur X est capable de gérer plusieurs applications sur plusieurs écrans à la fois. Pour que le serveur alloue les ressources disponibles (place mémoire) il est nécessaire d'établir une connexion avec lui.
  Note : Cette procédure utilise la primitive XOpenDisplay de X Windows et doit être appelée en premier lieu avant de réaliser une opération graphique sur l'écran. Si l'opération se passe mal le message <Impossible de créer une fenêtre sur ECRAN> est imprimé à l'écran.

+ INIT_XRLIB (P)

- argument :

- pré-conditions :
  OUVRIR_ECRAN déjà éxécutée
  INIT_XRLIB pas encore éxécutée

- résultat :

- post-conditions :

La librairie Xrlib est initialisée et il est donc possible à partir de ce moment d'utiliser le niveau Xrlib de X Windows. Ce niveau permet d'utiliser les structures de données attachés aux menus, éditeurs ..., il est donc indispensable d'initialiser cette librairie pour pouvoir manipuler ces concepts.

Note : Cette procédure utilise XrInit et doit être appelée en second lieu après OUVRIR_ECRAN. Si l'opération se passe mal le message <Impossible d'initialiser Xrlib > est imprimé à l'écran.

1.1.2 Fenêtre-texte

+ CREER_FENETRE_TEXT (F)

- arguments :

LARGEUR : largeur en caractères de la fenêtre-texte
HAUTEUR : hauteur en caractères de la fenêtre-texte
X : position en x (en pixels) où se trouvera la fenêtre-texte à l'écran
Y : position en y (en pixels) où se trouvera la fenêtre-texte à l'écran
PARENT : instance de la fenêtre dans laquelle on veut créer une fenêtre-texte
POLICE : police de caractères qui sera utilisée dans la fenêtre-texte lorsque du texte sera affiché dans la fenêtre
LARG_BORD : largeur du bord de la fenêtre-texte
SCROLL : 0 si on veut un défilement ligne par ligne
1 si on veut un défilement page par page

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
LARGEUR, HAUTEUR, X, Y > 0
PARENT doit être une instance d'une fenêtre obtenue soit par XCreateWindow soit par CREER_FENETRE
LARG_BORD ≥ 0

- résultat :

instance d'une fenêtre-texte qui devra être utilisée par la suite chaque fois que l'on voudra faire référence à cette fenêtre

- post-conditions :

Une fenêtre-texte conforme aux paramètres en entrée est créée sans être affichée. Si la police de caractères POLICE n'existe pas, le message <La police de caractères POLICE n'existe pas> est envoyé à l'écran et la police de caractères par défaut est utilisée. En outre cette fonction met en place la structure nécessaire pour redessiner automatiquement le contenu de la fenêtre-texte lorsqu'elle est cachée par une autre fenêtre et puis rendue visible (procédure DESSINER_TEXTE, cfr infra).

Note : Cette procédure utilise TextCreate, XrInput.

+ DESSINER_TEXTE (P)

- arguments :

FENETRE_INST : instance de la fenêtre
MESSAGE : message passé à la procédure
DONNEES : données passées à la procédure

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_INST est une instance d'une fenêtre obtenue soit par CREER_FENETRE_TEXT, soit par TextCreate et ne doit pas avoir été détruite
La fenêtre dans laquelle a été créée la fenêtre-texte doit être affichée à l'écran

- résultat :

- post-conditions :

Suivant le message passé à la procédure celle-ci réalisera les actions correspondantes à ce message. La seule valeur possible pour le message dans l'état actuel est la constante prédéfinie MSG_REDRAW. En cas de réception de ce message la procédure redessinera le texte contenu dans la fenêtre identifiée par FENETRE_INST.

Note : Cette procédure utilise XrInput et TextRedisplay.

+ AFFICHER_FENETRE_TEXT (P)

- arguments :

FENETRE_TEXT : instance de la fenêtre-texte que l'on veut afficher

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_TEXT doit être une instance d'une fenêtre obtenue soit par CREER_FENETRE_TEXT, soit par TextCreate et ne doit pas avoir été détruite.
La fenêtre dans laquelle a été créée la fenêtre-texte doit être affichée à l'écran

- résultat :

- post-conditions :

La fenêtre identifiée par FENETRE_TEXT est affichée à l'écran.

Note : Cette procédure utilise XMapWindow.

+ ECRIRE_TEXTE_LIGNE (P)

- arguments :

FENETRE_TEXT : instance de la fenêtre-texte dans laquelle on veut afficher la chaîne de caractères

CHAINE : chaîne de caractères que l'on veut afficher dans la fenêtre-texte

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée

INIT_XRLIB déjà éxécutée

FENETRE_TEXT doit être une instance d'une fenêtre obtenue soit par CREER_FENETRE_TEXT, soit par TextCreate et ne doit pas avoir été détruite

La fenêtre identifiée par FENETRE_TEXT doit être affichée à l'écran

- résultat :

- post-conditions :

La chaîne de caractères CHAINE est affichée dans la fenêtre-texte identifiée par FENETRE_TEXT à la suite du texte éventuellement déjà affiché dans cette fenêtre.

Note : Cette procédure utilise TextPutString.

+ EFFACER_FENETRE_TEXT (P)

- arguments :

FENETRE_TEXT : instance de la fenêtre-texte dont on veut effacer le contenu

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée

INIT_XRLIB déjà éxécutée

FENETRE_TEXT doit être une instance d'une fenêtre obtenue soit par CREER_FENETRE_TEXT, soit par TextCreate et ne doit pas avoir été détruite

La fenêtre identifiée par FENETRE_TEXT doit être affichée à l'écran

- résultat :

- post-conditions :

Le contenu de la fenêtre-texte identifiée par FENETRE_TEXT est effacé à l'écran.

Note : Cette procédure utilise TextClear.

+ **DETRUIRE_FENETRE_TEXT (P)**

- arguments :
  FENETRE_TEXT : instance de la fenêtre-texte que l'on veut détruire

- pré-conditions :
  OUVRIR_ECRAN déjà éxécutée
  INIT_XRLIB déjà éxécutée
  FENETRE_TEXT doit être une instance d'une fenêtre obtenue soit par CREER_FENETRE_TEXT, soit par TextCreate
  La fenêtre identifiée par FENETRE_TEXT doit être affichée à l'écran

- résultat :

- post-conditions :
  La fenêtre-texte identifiée par FENETRE_TEXT est détruite et effacée de l'écran.
  Note : Cette procédure utilise TextDestroy.

+ **REAFFICHER_FENETRE_TEXT (P)**

- arguments :
  FENETRE_TEXT : instance de la fenêtre-texte dont on veut réafficher le contenu

- pré-conditions :
  OUVRIR_ECRAN déjà éxécutée
  INIT_XRLIB déjà éxécutée
  FENETRE_TEXT doit être une instance d'une fenêtre obtenue soit par CREER_FENETRE_TEXT, soit par TextCreate
  La fenêtre identifiée par FENETRE_TEXT doit être affichée à l'écran

- résultat :

- post-conditions :
  Le contenu de la fenêtre-texte identifiée par FENETRE_TEXT est réaffiché à l'écran.
  Note : Cette procédure utilise TextRedisplay.

1.1.3 Fenêtre

+ CREER_FENETRE (F)

- arguments :
  PARENT_INST : instance du père de la fenêtre que l'on va créer
  POSX : position en x (en pixels) où l'on veut que la fenêtre soit affichée à l'écran
  POSY : position en y (en pixels) où l'on veut que la fenêtre soit affichée à l'écran
  LARGEUR : largeur (en pixels) de la fenêtre qui sera créée
  HAUTEUR : hauteur (en pixels) de la fenêtre qui sera créée
  LARG_BORD : largeur du bord c'est-à-dire du cadre qui entoure la fenêtre qui sera créée

- pré-conditions :
  OUVRIR_ECRAN déjà éxécutée
  INIT_XRLIB déjà éxécutée

- résultat : instance de la fenêtre créée

- post-conditions :
  Une fenêtre conforme aux paramètres en entrée est créée. En outre cette procédure met en place la structure nécessaire pour redessiner automatiquement les éditeurs de la fenêtre c'est-à-dire appeler DESSINER_EDITEUR dans les cas où la fenêtre est cachée par une autre fenêtre et puis rendue visible. De plus cette procédure crée et attache un curseur à la fenêtre, et enregistre la fenêtre dans les tables de Xrlib.
  Note : Cette procédure utilise XCreateWindow, XrInput, XDefineCursor.

+ DESSINER_EDITEUR (P)

- arguments :
  FENETRE_INST : instance de la fenêtre
  MESSAGE : message passé à la procédure
  DONNEES : données passées à la procédure

- pré-conditions :
  OUVRIR_ECRAN déja éxécutée
  INIT_XRLIB déjà éxécutée
  FENETRE_INST est une instance d'une fenêtre obtenue soit par CREER_FENETRE, soit par XCreateWindow

- résultat :

- post-conditions :

Suivant le message passé à la procédure celle-ci réalisera les actions correspondantes à ce message. La seule valeur possible pour le message dans l'état actuel est la constante prédéfinie MSG_REDRAW. En cas de réception de ce message la procédure redessinera une partie de la fenêtre identifiée par FENETRE_INST. La partie de la fenêtre qu'il faut redessiner est donnée dans le paramètre DONNEES.

Note : Cette procédure utilise XrEditor, XrInput.

Cette fonction ainsi que DESSINER_TEXTE (cfr 1.1.2) est nécessaire du fait que X Windows ne redessine pas automatiquement les fenêtres lorsqu'elles sont cachées par une autre fenêtre et puis rendues visibles par la suite (cfr chapitre 2 section 7). Ces fonctions sont donc appelées automatiquement lorsqu'une fenêtre qui était cachée par une autre fenêtre est rendue visible. La fonction DESSINER_EDITEUR s'occupe de redessiner les editeurs attachés à la fenêtre, tandis que DESSINER_TEXTE s'occupe de redessiner le texte qui se trouve dans la fenêtre.

+ AFFICHER_FENETRE (P)

- arguments :

FENETRE_INST : instance de la fenêtre que l'on veut afficher à l'écran

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée

- résultat :

- post-conditions :

La fenêtre identifiée par FENETRE_INST est affichée à l'écran.
Note : Cette procédure utilise XMapWindow.

+ DEMAPPER_FENETRE (P)

- arguments :

FENETRE_INST : instance de la fenêtre que l'on veut effacer de l'écran

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée

- résultat :

- post-conditions :

La fenêtre identifiée par FENETRE_INST est effacée de l'écran sans être tuée c'est-à-dire qu'il est toujours possible de réafficher la fenêtre avec la procédure AFFICHER_FENETRE (cfr supra).

Note : Cette procédure utilise XUnmapWindow.

+ ECRIRE_TEXTE (P)

- arguments :

FENETRE_INST : instance de la fenêtre où on veut écrire le texte
POSX : position en x (en pixels) relative à la fenêtre où l'on veut écrire le texte
POSY : position en y (en pixels) relative à la fenêtre où l'on veut écrire le texte
TEXTE : chaîne de caractères que l'on veut écrire à l'écran
POLICE : police de caractères que l'on veut utiliser

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_INST doit être l'instance d'une fenêtre obtenue en utilisant la procédure CREER_FENETRE où XCreateWindow

- résultat :

- post-conditions :

La chaîne de caractères TEXTE est affichée dans la fenêtre identifiée par FENETRE_INST à la position (XPOS, YPOS) relative à la fenêtre avec la police de caractères POLICE. Si la police de caractères POLICE n'existe pas le message d'erreur <La police de caractères POLICE n'existe pas> est imprimé à l'écran et la police de caractères par défaut est utilisée.

Note : Cette procédure utilise XOpenFont, XText, XCloseFont.

+ LARGEUR_FENETRE (F)

- arguments :

FENETRE_INST : instance de la fenêtre dont on veut connaître la largeur

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_INST doit être une instance de fenêtre obtenue soit par XCreateWindow soit par CREER_FENETRE et ne doit pas avoir été détruite

- résultat :

largeur (en pixels) de la fenêtre identifiée par FENETRE_INST

- post-conditions :

La largeur de la fenêtre identifiée par FENETRE_INST est renvoyée.

Note : Cette procédure utilise XQueryWindow.

+ HAUTEUR_FENETRE (F)

- arguments :

FENETRE_INST : instance de la fenêtre dont on veut connaître la hauteur

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée

INIT_XRLIB déjà éxécutée

FENETRE_INST doit être une instance de fenêtre obtenue soit par XCreateWindow soit par CREER_FENETRE et ne doit pas avoir été détruite

- résultat :

hauteur (en pixels) de la fenêtre identifiée par FENETRE_INST

- post-conditions :

La hauteur de la fenêtre identifiée par FENETRE_INST est renvoyée.

Note : Cette procédure utilise XQueryWindow.

+ CHANGER_TAILLE_FENETRE (P)

- arguments :

FENETRE_INST : instance de la fenêtre dont on veut changer la taille

LARGEUR : largeur (en pixels) que l'on veut donner à la fenêtre

HAUTEUR : hauteur (en pixels) que l'on veut donner à la fenêtre

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée

INIT_XRLIB déjà éxécutée

FENETRE_INST doit être une instance d'une fenêtre obtenue soit par CREER_FENETRE, soit par XCreateWindow et ne doit pas avoir été détruite.

LARGEUR, HAUTEUR > 0

- résultat :

- post-conditions :

La fenêtre identifiée par FENETRE_INST a une largeur égale à LARGEUR et une hauteur égale à HAUTEUR.

Note : Cette procédure utilise XChangeWindow.

+ **CHANGER_POSITION_FENETRE (P)**

- arguments :

FENETRE_INST : instance de la fenêtre dont on veut changer la position
X : position en x (en pixels) à laquelle on veut déplacer la fenêtre identifiée par FENETRE_INST à l'écran
Y : position en y (en pixels) à laquelle on veut déplacer la fenêtre identifiée par FENETRE_INST à l'écran

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_INST doit être une instance d'une fenêtre obtenue soit par CREER_FENETRE, soit par XCreateWindow et ne doit pas avoir été détruite.
$0 \leq X \leq$ largeur (ecran)
$0 \leq Y \leq$ hauteur (ecran)

- résultat :

- post-conditions :

La fenêtre identifiée par FENETRE_INST est positionnée à l'écran en (X, Y)
Note : Cette procédure utilise XMoveWindow.

1.1.4 Barre de titre

+ **CREER_BARRE_TITRE**

- arguments :

FENETRE_INST : instance de la fenêtre dans laquelle on veut créer la barre de titre
TITRE : chaîne de caractères qui représente le titre qui sera mis dans la barre de titre
POLICE : police de caractères que l'on veut utiliser
GADJ_BOITE1_R : chaîne de caractères que l'on veut mettre dans la boîte à sélection gauche
GADJ_BOITE2_R : chaîne de caractères que l'on veut mettre dans la boîte à sélection droite

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_INST doit être l'identificateur d'une fenêtre obtenue en utilisant la procédure CREER_FENETRE où XCreateWindow
$0 \leq$ longueur (GADJ_BOITE1_R), longueur (GADJ_BOITE2_R) ≤ 1

- résultat :

instance de la barre de titre créée qui devra être utilisée par la suite si l'on veut faire référence à la barre de titre

- post-conditions :

Une barre de titre conforme aux paramètres en entrée est créée en-dessous du cadre supérieur de la fenêtre FENETRE_INST. Dans le cas où une chaîne de caractères pour les boîtes a une longueur égale à zéro, la boîte n'est pas créée dans la barre de titre. Si la police de caractères POLICE n'existe pas, le message d'erreur <La police de caractères POLICE n'existe pas> est imprimé à l'écran et la police de caractères par défaut est utilisée.

Note : Cette procédure utilise XrSetRect, XOpenFont, XrTitleBar

+ AFFICHER_BARRE_TITRE

- arguments :

BARRE_TITRE_INST : instance de la barre de titre que l'on veut afficher à l'écran

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée

- résultat :

- post-conditions :

La barre de titre identifiée par BARRE_TITRE_INST est affichée à l'écran dans la fenêtre où elle a été créée.

Note : Cette procédure utilise XrTitleBar

1.1.5 Gestion de la file d'événements associée à l'application

+ ENVOYER_MANAGER (P)

- arguments :

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée

- résultat :

- post-conditions :

Les événements qui ont été générés mais pas encore traités sont supprimés et traités par le gestionnaire des événements de X Windows.

Note : Cette procédure utilise XFlush.

1.1.6 Editeur radio

+ CREER_RADIO_BOUT (F)

- arguments :

FENETRE_INST : instance de la fenêtre dans laquelle on veut créer l'éditeur radio
CHAMP_R : tableau des chaînes de caractères qui seront placées à côté des différents boutons du radiobutton
NB_CHAMP : nombre effectif de champs du radiobutton
NB_COLS : nombre de colonnes sur lesquelles on veut que les boutons du radiobutton soient affichés
POLICE_TITRE : police utilisée pour la barre de titre de la fenêtre dans laquelle sera créée le radiobutton. Le radiobutton sera affiché en dessous de la barre de titre
POLICE_RADIO : police utilisée pour les labels qui sont placés à côté des boutons du radiobutton

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_INST doit être l'identificateur d'une fenêtre obtenue en utilisant la procédure CREER_FENETRE où XCreateWindow
CHAMP_R contient au maximum 20 chaînes de caractères
$0 \leq NB\_CHAMP < 20$

- résultat : instance de l'éditeur radio créé

- post-conditions :

Un radiobutton avec NB_CHAMP champs et NB_COLS colonnes est créé dans la fenêtre identifiée par FENETRE_INST en dessous de la barre de titre. Les labels CHAMP_R[1.. NB_CHAMP] ont été affichés en utilisant la police de caractères POLICE_RADIO. Si la police de caractères POLICE_RADIO ou POLICE_TITRE n'existe pas le message d'erreur <La police de caractères POLICE_RADIO (POLICE_TITRE) n'existe pas> est imprimé à l'écran et la police de caractères par défaut est utilisée.

Note : Cette procédure utilise XOpenFont, XQueryWindow, XrOffsetRect, XrRadioButton.

+ **AFFICHER_RADIO_BOUTON (P)**

- arguments :
  RADIO_BOUT_INST : instance du radiobutton que l'on veut afficher à l'écran

- pré-conditions :
  OUVRIR_ECRAN déjà éxécuté
  INIT_XRLIB déjà éxécuté
  La fenêtre dans laquelle le radiobutton a été créé doit être affichée à l'écran.

- résultat :

- post-conditions :
  Le radiobutton identifié par RADIO_BOUT_INST est affiché dans la fenêtre où il a été créé.
  Note : Cette procédure utilise XrRadioButton.

+ **OBTENIR_HAUT_RADIO_BOUT (F)**

- arguments :
  CHAMP : tableau des chaînes de caractères qui constituent les labels des boutons du radiobutton dont on veut obtenir la hauteur
  NB_CHAMP : nombre effectif de boutons du radiobutton
  NB_COLS : nombre de colonnes du radiobutton dont on veut obtenir la hauteur
  POLICE_RADIO : police utilisée pour les labels qui sont placés à côté des boutons du radiobutton

- pré-conditions :
  OUVRIR_ECRAN déjà éxécutée
  INIT_XRLIB déjà éxécutée
  CHAMP contient au maximum 20 chaînes de caractères
  $0 \leq NB\_CHAMP \leq 20$

- résultat :
hauteur du radiobutton

- post-conditions :
  La hauteur qu'aurait le radiobutton s'il était créé avec les paramètres donnés est renvoyée.
  Note : Cette procédure utilise XOpenFont, XrRadioButton.

1.1.9 Menu

+ CREER_MENU (F)

- arguments :

  MENU_ID : identificateur c'est-à-dire nombre qui identifie le menu parmi les autres menus

  ITEM_R : tableau des chaînes de caractères qui constitueront les différentes sélections possibles du menu

  NB_ITEMS : nombre effectif d'items du menu

  MENU_TITRE : chaîne de caractères qui constituera le titre du menu

  POLICE : police utilisée pour les items et le titre du menu

- pré-conditions :

  OUVRIR_ECRAN déjà éxécuté

  INIT_XRLIB déjà éxécuté

  ITEM_R contient au maximum 20 chaînes de caractères

  $0 < NB\_ITEMS \leq 20$

- résultat :

  instance du menu créé qui devra être utilisée par la suite chaque fois que l'on voudra faire référence au menu

- post-conditions :

  Un menu avec NB_ITEMS items est créé avec les sélections ITEM_R[1 .. NB_ITEMS] et le titre MENU_TITRE en utilisant pour tous ces composants la police de caractères POLICE. Si la police de caractères POLICE n'existe pas le message d'erreur <La police de caractères POLICE n'existe pas> est imprimé à l'écran et la police de caractères par défaut est utilisée.

  Note : Cette procédure utilise XOpenFont, XrMenu.

+ ATTACHER_SM_MENU (P)

- arguments :

  SOUS_MENU_INST : instance du sous-menu que l'on veut attacher au menu

  MENU_INST : instance du menu auquel on veut rattacher le sous-menu

  INDEX_ITEM : numéro de l'item (numérotation à partir de 0) auquel on veut attacher le sous-menu

- pré-conditions :

  OUVRIR_ECRAN déjà éxécutée

  INIT_XRLIB déjà éxécutée

  SOUS_MENU_INST et MENU_INST doivent être des instances de menu obtenues en utilisant la fonction CREER_MENU ou XrMenu.

- résultat :

- post-conditions :

Le sous-menu identifié par SOUS_MENU_INST est attaché au menu identifié par MENU_INST à l'endroit où se trouve l'item de numéro INDEX_ITEM. On pourra sélectionner une commande du sous-menu en se positionnant sur la flèche qui se trouve près de l'item de numéro INDEX_ITEM du menu identifié par MENU_INST.

Note : Cette procédure utilise XrMenu.

+ ATTACHER_MENU_FEN (P)

- arguments :

MENU_INST : instance du menu que l'on veut attacher à la fenêtre
FENETRE_INST : instance de la fenêtre à laquelle on veut attacher le menu

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
MENU_INST doit être une instance de menu obtenue en utilisant la fonction CREER_MENU ou XrMenu et ne pas avoir été détruit
FENETRE_INST doit être une instance de fenêtre obtenue soit par XCreateWindow soit par CREER_FENETRE et ne pas avoir été détruite

- résultat :

- post-conditions :

Le menu identifié par MENU_INST est attaché à la fenêtre identifiée par FENETRE_INST.

+ ACTIVER_ITEM_MENU (P)

- arguments :

MENU_INST : instance du menu dont on veut activer un item c'est-à-dire une sélection
NUMERO_ITEM : numéro de l'item (compté à partir de zéro) que l'on veut activer

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
MENU_INST est l'instance d'un menu qui est créé et non détruit
NUMERO_ITEM doit être un numéro d'item valide pour le menu identifié par MENU_INST

- résultat :

- post-conditions :

L'item numéro NUMERO_ITEM du menu identifié par MENU_INST est activé c'est-à-dire que si l'item n'était pas sélectionnable, il le devient.

Note : Cette procédure utilise XrMenu.

+ DESACTIVER_ITEM_MENU (P)

- arguments :

MENU_INST : instance du menu dont on veut désactiver un item c'est-à-dire une sélection

NUMERO_ITEM : numéro de l'item (compté à partir de zéro) que l'on veut désactiver

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée

INIT_XRLIB déjà éxécutée

MENU_INST est une instance d'un menu créé et non détruit

NUMERO_ITEM doit être un numéro d'item valide pour le menu identifié par MENU_INST

- résultat :

- post-conditions :

L'item numéro NUMERO_ITEM du menu identifié par MENU_INST est désactivé c'est-à-dire que si l'item était sélectionnable, il devient insélectionnable.

Note : Cette procédure utilise XrMenu.

1.1.8 Attacher les types d'événements entrées standard à une fenêtre

+ TYPE_ENTREES_STAND (P)

- arguments :

FENETRE_INST : instance de la fenêtre a laquelle on veut affecter le type d'entrées standard

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée

INIT_XRLIB déjà éxécutée

FENETRE_INST doit être une instance de fenêtre obtenue soit par XCreateWindow soit par CREER_FENETRE

- résultat :

- post-conditions :

La fenêtre identifiée par FENETRE_INST se voit affecter le type d'entrées standard c'est-à-dire que cette fenêtre peut recevoir un certain nombre d'événements. Ces événements sont (cfr doc X Windows) ButtonPressed, ButtonReleased, KeyPressed, KeyReleased, ExposeWindow, ExposeRegion, LeaveWindow. Cela signifie par exemple que puisque ButtonReleased est présent lorsque la souris est relachée dans la fenêtre cet événement sera pris en compte et mis dans la file d'événements pour être traité.

Note : Cette procédure utilise XrInput.

1.1.9 Boîte à messages

+ CREER_BOITE_MESSAGE (F)

- arguments :

FENETRE_INST : instance de la fenêtre dans laquelle on veut créer une boîte à messages

BOITE_MESSAGE_ID : identificateur c'est-à-dire nombre qui identifie la boîte à messages par rapport aux autres boîtes à messages

PIX_MAP : instance d'un pixmap c'est-à-dire un petit dessin qui figurera dans la boîte à messages

HAUT_PIX : hauteur (en pixels) du pixmap

LARG_PIX : largeur (en pixels) du pixmap

MESSAGE : chaîne de caractères qui constituera le message figurant dans la boîte à messages

BOUTON : tableau des chaînes de caractères constituant les labels associés aux boutons de l'éditeur à boutons de la boîte à messages

NB_BOUTONS : nombre effectif de boutons de l'éditeur à boutons

POS_X_FEN : la boîte a messages est placée au centre de la fenêtre si possible, c'est-à-dire que le milieu en x de la boîte à messages coïncide avec le milieu en x de la fenêtre et le milieu en y de la boîte à messages coïncide avec le milieu en y de la fenêtre, sauf dans le cas où elle sortirait de l'écran. Dans ce cas, la boîte à messages est positionnée de manière telle qu'elle ne sorte pas de l'écran. POS_X_FEN représente l'incrément où le décrément que l'on veut ajouter ou retirer à la position en x de la boîte à messages calculé comme dit ci-dessus.

POS_Y_FEN : incrément ou décrément pour la position en y de la boite à messages (cfr POS_X_FEN)

POLICE : police de caractères utilisée pour les labels des boutons de l'éditeur à boutons et pour le message

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée

INIT_XRLIB déjà éxécutée

FENETRE_INST doit être une instance de fenêtre obtenue soit par XCreateWindow soit par CREER_FENETRE et être affichée

- résultat :

- post-conditions :

Une boîte à messages respectant les paramètres en entrée est créée et affichée. Si la police de caractères POLICE n'existe pas le message d'erreur <La police de caractères POLICE n'existe pas> est imprimé à l'écran et la police de caractères par défaut est utilisée.

Note : Cette procédure utilise XQueryWindow, XrMessageBox, XrInput.

1.1.10 Les graphiques

+ CREER_PIXMAP_FEN (F)

- arguments :

FENETRE_INST : instance de la fenêtre dont on veut réaliser un pixmap
COORD_X : coordonnée en x (en pixels) dans la fenêtre à partir de laquelle le pixmap va être créé
COORD_Y : coordonnée en y (en pixels) dans la fenêtre à partir de laquelle le pixmap va être créé
HAUTEUR : hauteur du pixmap
LARGEUR : largeur du pixmap

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_INST doit être une instance de fenêtre obtenue soit par XCreateWindow soit par CREER_FENETRE et être affichée
$0 \leq$ COORD_X, LARGEUR $\leq$ largeur (FENETRE_INST)
$0 \leq$ COORD_Y, HAUTEUR $\leq$ hauteur (FENETRE_INST)

- résultat :

instance de pixmap qui devra être utilisée par la suite lorsqu'on voudra faire référence au pixmap

- post-conditions :

Un pixmap c'est-à-dire dans ce cas un dessin de la fenêtre à partir de la coordonnée en (X, Y) dans la fenêtre et de hauteur HAUTEUR et de largeur LARGEUR est créé. Cette procédure réalise donc une copie graphique d'une partie de la fenêtre.
Note : Cette procédure utilise XPixmapSave.

+ LIBERER_PIXMAP (P)

- argument :

PIX_MAP : instance du pixmap à libérer en mémoire

- pré-conditions :
  OUVRIR_ECRAN déjà éxécutée
  INIT_XRLIB déjà éxécutée
  PIX_MAP est une instance de pixmap obtenue soit par CREER_PIXMAP_FEN, soit par XPixmapSave et non encore libéré

- résultat :

- post-conditions :
  Le pixmap identifié par PIX_MAP est libéré en mémoire. En effet les pixmap sont stockés en mémoire et après utilisation il est préférable de les détruire pour ne pas occuper inutilement de la place en mémoire.
  Note : Cette procédure utilise XFreePixmap.

+ LIRE_PIXMAP (F)

- argument :
  NOMFICH : nom du fichier dans lequel se trouve les informations sur le pixmap

- pré-condition :

- résultat :
  instance du pixmap dont les informations se trouvent dans le fichier

- post-condition :
  Si le fichier de nom NOMFICH n'existe pas la fonction retourne une instance de pixmap par défaut dont les informations auront été incluses dans le fichier "XMYINCL.H". Dans le cas contraire l'instance du pixmap dont les informations se trouvent dans le fichier de nom NOMFICH est renvoyée.

1.1.11 Les polices de caractères

+ HAUTEUR_CAR (F)

- arguments :
  POLICE : police de caractères dont on désire connaître la hauteur des caractères

- pré-conditions :
  OUVRIR_ECRAN déjà éxécutée
  INIT_XRLIB déjà éxécutée

- résultat :
  hauteur des caractères de la police de caractères POLICE

- post-conditions :

La hauteur maximale des caractères de la police de caractères POLICE est renvoyée. Si la police de caractères POLICE n'existe pas, le message d'erreur <La police de caractères POLICE n'existe pas> est imprimé à l'écran et une hauteur égale à 0 est renvoyée.

Note : Cette procédure utilise XOpenFont, XCloseFont.

+ LARGEUR_CAR (F)

- arguments :

POLICE : police de caractères dont on veut connaître la largeur des caractères

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée

- résultat :

largeur des caractères de la police de caractères POLICE

- post-conditions :

La hauteur maximale des caractères de la police de caractères POLICE est renvoyée. Si la police de caractères POLICE n'existe pas, le message d'erreur <La police de caractères POLICE n'existe pas> est imprimé à l'écran et une largeur égale à 0 est renvoyée.

Note : Cette procédure utilise XOpenFont, XCloseFont.

1.1.12 Chaîne de caractères

+ LONG_CHAINE (P)

- arguments :

CHAINE : chaîne de caractères dont on veut connaître la longueur
POLICE : police de caractères utilisée pour CHAINE
ESP_CAR : largeur en pixels qu'il y a entre les caractères
ESP_BLANC : largeur du caractère blanc dans la police de caractères POLICE

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
ESP_CAR, ESP_BLANC > 0

- résultat :

longueur de la chaîne de caractères CHAINE en utilisant la police de caractères POLICE.

- post-conditions :

La longueur de la chaîne de caractères CHAINE est renvoyée. Si la police de caractères POLICE n'existe pas le message d'erreur <La police de caractères POLICE n'existe pas> est envoyé à l'écran et une longueur égale à 0 est retournée.

Note : Cette procédure utilise XOpenFont, XCloseFont, XrStringWidth.

1.1.13 File d'attente pour les fenêtres text

+ MANAGER_TEXT (P)

- arguments :

FENETRE_TEXT : instance de la fenêtre-texte dont on veut vider la file d'événements

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée
FENETRE_TEXT doit être une instance d'une fenêtre obtenue soit par CREER_FENETRE_TEXT, soit par TextCreate
La fenêtre identifiée par FENETRE_TEXT doit être affichée à l'écran.

- résultat :

- post-conditions :

La file d'événements de la fenêtre identifiée par FENETRE_TEXT est vidée.
Note : Cette procédure utilise TextFlush.

1.1.14 Gestion des événements

+ TRAITER_EVENEMENTS (P)

- argument :

- pré-conditions :

OUVRIR_ECRAN déjà éxécutée
INIT_XRLIB déjà éxécutée

- résultat :

- post-conditions :

Cette procédure est chargée de traiter tous les événements générés par l'utilisateur de l'application. Cette procédure est donc spécifique à l'application. Elle est constituée d'une boucle dans laquelle on lit et on traite continuellement des événements. Seul un squelette de gestion des événements est proposé. Ce squelette doit être rempli pour une application particulière.

Cette procédure est décomposée en sous-procédures chargées chacune de traiter une classe d'événements différente. Ces sous-procédures sont elles-mêmes décomposées en procédures de traitement des événements plus spécifiques.

Ces sous-procédures sont les suivantes :

* TRAITER_EDITEUR : traite les événements générés par les éditeurs. Cette procédure est décomposée en sous-procédures traitant les événements suivant le type de l'éditeur qui les a générés. Il s'agit essentiellement de réaliser les actions correspondantes aux différents événements qui peuvent être reçus de chaque éditeur.
  - TRAITER_PUSH_BOUT : traite les événements générés par l'éditeur à boutons (pushbutton)
  - TRAITER_BARRE_TITRE : traite les événements générés par la barre de titre (titlebar)
  - TRAITER_TXTEDIT : traite les événements générés par l'éditeur de texte (pageedit)
  - TRAITER_RADIO_BOUT : traite les événements générés par l'éditeur radio (radiobutton)

* TRAITER_MENU : traite les événements générés par les menus. Cette procédure est décomposée en autant de sous-procédures qu'il y a de menus avec des identificateurs de menus différents. Chaque sous-procédure est chargée de réaliser les actions correspondantes aux items du menu dont elle est chargée de traiter les événements.

* TRAITER_BOITE_MESSAGE : traite les événements générés par les boîtes à messages (messagebox). Cette procédure est décomposée en autant de sous-procédures qu'il y a de boîtes à messages avec des identificateurs différents. Chaque sous-procédure est chargée de traiter les événements de la boîte à messages dont elle est responsable.

A1.2 Fichier PRIMXED.C

1.2.1 Editeur de texte

+ CREER_TXTEDIT (F)

- arguments :

  FENID : instance de la fenêtre dans laquelle doit être créé l'éditeur de texte
  LARG_EDIT : largeur d'édition en nombre de colonnes
  HAUT_EDIT : hauteur d'édition en nombre de lignes
  POLICE_EDIT : police de caractères

- pré-condition :

FENID identifie une fenêtre existante

POLICE_EDIT est une police de caractères proportionnelle

- résultat :

instance de l'éditeur de texte créé qui devra être utilisé par la suite lorsqu'il faut faire référence à cet éditeur de texte

- post-conditions :

Un éditeur de texte permettant le gestion d'un texte composé de plusieurs lignes est créé. Il est rattaché à la fenêtre FENID, est composé de LARG_EDIT colonnes et de HAUT_EDIT lignes de caractères.

Lorsqu'il n'est pas possible d'utiliser la police de caractères passée en paramètre, l'éditeur utilise une police définie par défaut dans le fichier "XMYINCL.H".

La taille du buffer d'édition (de la "zone d'édition") est définie en terme de bloc de caractères ou "unité d'édition". La taille de cette "unité d'édition" est définie en constante dans le fichier cité plus haut. Lorsque la taille du buffer d'édition n'est pas assez grande pour le texte entré au terminal, l'éditeur augmente la dite zone d'édition d'une "unité d'édition".

L'éditeur de texte sera positionné au bas de la fenêtre FENID.

+ AFFICHER_TXTEDIT (P)

- argument :

TEXT_EDIT_NOM : instance de l'éditeur de texte à afficher

- pré-condition :

TEXT_EDIT_NOM identifie un éditeur existant

- résultat :

- post-conditions :

L'éditeur identifié par TEXT_EDIT_NOM est affiché à l'écran

+ ACTIVER_TXTEDIT (P)

- argument :

TEXT_EDIT_NOM : instance de l'éditeur de texte à rendre "sensible"

- pré-condition :

TEXT_EDIT_NOM identifie un éditeur existant

L'éditeur identifié par TEXT_EDIT_NOM est affiché à l'écran

- résultat :

- post-conditions :
L'éditeur identifié par TEXT_EDIT_NOM est "sensible"

+ DEACTIVER_TXTEDIT (P)

- argument :
TEXT_EDIT_NOM : instance de l'éditeur de texte à rendre "insensible"

- pré-condition :
TEXT_EDIT_NOM identifie un éditeur existant
L'éditeur identifié par TEXT_EDIT_NOM est affiché à l'écran

- résultat :

- post-conditions :
L'éditeur identifié par TEXT_EDIT_NOM est "insensible"

+ TRAITE_TXTEDIT (P)

Cette fonction traite tous les événements ayant rapport à un "éditeur" éditeur de texte.

- argument :
INPUT : événement à traiter

- pré-conditions :
INPUT est un événement qui vient d'un "éditeur" éditeur de texte
L'éditeur est "sensible"

- résultat :

- post-conditions :
L'événement INPUT a été traité et enlevé de la file d'événements

+ VOIR_CONT_BUF (F)

- argument :
TEXT_EDIT_NOM : instance de l'éditeur de texte dont on veut avoir une copie du buffer

- pré-condition :
TEXT_EDIT_NOM identifie un éditeur qui existe

- résultat :
copie du contenu du buffer d'édition

- post-condition :

Une copie du buffer d'édition de l'éditeur de texte identifié par TEXT_EDIT_NOM est renvoyé

+ **INIT_CONT_BUF**

- argument :

TEXT_EDIT_NOM : instance de l'éditeur de texte

- pré-condition :

TEXT_EDIT_NOM identifie un éditeur qui existe

- résultat :

- post-conditions :

Le buffer d'édition de l'éditeur de texte identifié par TEXT_EDIT_NOM la une taille d'une "unité d'édition".

+ **EXPANS_BUF (P)**

- argument :

TEXT_EDIT_NOM : instance de l'éditeur de texte dont on veut augmenter la taille du buffer d'édition

- pré-condition :

TEXT_EDIT_NOM identifie un éditeur de texte qui existe

- résultat :

- post-conditions :

La taille du buffer d'édition de l'éditeur de texte identifié par TEXT_EDIT_NOM est augmenté d'une "unité d'édition".

+ **CHANGE_CONT_BUF (P)**

- arguments :

TEXT_EDIT_NOM : instance de l'éditeur de texte
TEXTE : texte à mettre dans le buffer d'édition

- pré-conditions :

TEXT_EDIT_NOM identifie un éditeur de texte qui existe

- résultat :

- post-conditions :

Le contenu du buffer d'édition de l'éditeur de texte identifié par TEXT_EDIT_NOM est TEXTE.

1.2.2 L'éditeur à boutons

+ CREER_ED_BOUT (F)

- arguments :

FENETRE_INST : instance de la fenêtre dans laquelle l'éditeur à boutons doit être créé
CHAMP : tableau des chaînes de caractères qui constitueront les labels des boutons de l'éditeur à boutons
NB_CHAMP : nombre effectif de boutons de l'éditeur à boutons
NB_COLS : nombre de colonnes de l'éditeur à boutons
POLICE_TITRE : police de caractères utilisée pour le titre
POLICE_ED_BOUT : police de caractères utilisée pour les labels des boutons

- pré-condition :

FENETRE_INST identifie une fenêtre existante

- résultat :

instance de l'éditeur à boutons qui devra être utilisée par la suite lorsque l'on voudra faire référence à cet éditeur à boutons

- post-conditions :

Un éditeur à boutons avec un nombre NB_CHAMPS de boutons rangés en NB_COLS colonnes est créé dans la fenêtre identifiée par FENETRE_INST en dessous de la barre de titre. La police de caractères de la barre de titre doit être donnée pour pouvoir placer l'éditeur à boutons en dessous de la barre de titre de la fenêtre. Si une des deux polices de caractères POLICE_TITRE ou POLICE_ED_BOUT n'existe pas le message < La police de caractères POLICE_TITRE (POLICE_ED_BOUT) > est affiché à l'écran et la police de caractères par défaut est utilisée.

+ AFFICHER_ED_BOUTON (P)

- argument :

ED_BOUT_INST : instance de l'éditeur à boutons à afficher

- pré-condition :

ED_BOUT_INST identifie un éditeur à boutons existant

- résultat :

- post-conditions :

L'éditeur à boutons identifié par ED_BOUT_INST est affiché dans la fenêtre dans laquelle il a été créé

+ ACTIV_ED_BOUTON (P)

- argument :

ED_BOUT_INST : instance de l'éditeur à boutons à rendre "sensible"

- pré-condition :

ED_BOUT_INST identifie un éditeur à boutons existant et affiché

- résultat :

- post-conditions :

L'éditeur à boutons identifié par ED_BOUT_INST est "sensible"

+ DEACT_ED_BOUTON (P)

- argument :

ED_BOUT_INST : instance de l'éditeur à boutons à rendre "insensible"

- pré-condition :

ED_BOUT_INST identifie un éditeur à boutons existant et affiché

- résultat :

- post-conditions :

L'éditeur à boutons identifié par ED_BOUT_INST est "insensible"

+ ACTIV_IND_ED_BOUTON (P)

- arguments :

ED_BOUT_INST : instance de l'éditeur à boutons
NO_CHAMP : identificateur du bouton particulier de l'éditeur à boutons à rendre "sensible"

- pré-conditions :

ED_BOUT_INST identifie un éditeur à boutons existant et affiché
NO_CHAMP est l'identificateur d'un bouton existant dans l'éditeur à boutons

- résultat :

- post-conditions :

Le bouton identifié par NO_CHAMP est "sensible" dans l'éditeur à boutons identifié par ED_BOUT_INST

+ DEACT_IND_ED_BOUTON (P)

- arguments :

ED_BOUT_INST : instance de l'éditeur à boutons
NO_CHAMP : identificateur d'un bouton particulier de l'éditeur à boutons à "rendre insensible"

- pré-condition :

ED_BOUT_INST identifie un éditeur à boutons existant et affiché
NO_CHAMP est l'identificateur d'un bouton existant dans l'éditeur à boutons

- résultat :

- post-conditions :

Le bouton identifié par NO_CHAMP est "insensible" dans l'éditeur à boutons identifié par ED_BOUT_INST

+ OBTENIR_HAUT_ED_BOUT (F)

- arguments :

CHAMP : tableau qui contient les chaînes de caractères affichés à côté des boutons de l'éditeur à boutons dont on veut obtenir la hauteur
NB_CHAMP : nombre effectif de boutons de l'éditeur à boutons
NB_COLS : nombre de colonnes de l'éditeur à boutons
POLICE_ED_BOUT : police de caractères utilisée pour les boutons de l'éditeur à boutons

- pré-condition :

CHAMP contient au maximum 20 chaînes de caractères
$0 < NB\_CHAMP \leq 20$

- résultat :

hauteur (en pixels) de l'éditeur à boutons

- post-condition :

La hauteur qu'aurait un éditeur à boutons composé de NB_CHAMP boutons avec les labels respectifs CHAMP[1.. NB_CHAMP] sur NB_COLS colonnes et utilisant la police de caractères POLICE_ED_BOUT est renvoyée.

Annexe 2

Cette annexe reprend les procédures LE_LISP qui ont été ajoutées (A) ou modifiées (M) par rapport à l'ancienne version de SACSO. Les modifications sont reprises par fichier.

Les procédures qui ont été modifiées sont essentiellement les anciennes procédures responsables de la gestion du multifenêtrage sur écran alphanumérique. Toutefois, certaines procédures ont vues leur nombre de paramètres changer et tous les appels à ces procédures ont donc dû être modifiés. Nous ne reprenons que les fichiers de SACSO qui ont été modifiés ou ajoutés.

A2.1 Fichier Xdefextern.ll

Ce fichier contient toutes les déclarations des fonctions C en LE_LISP. Les procédures définies en C (cfr annexe 1) sont donc appelables directement depuis LE_LISP. Ces fonctions en été définies en LE_LISP grâce à la fonction LE_LISP defextern. Pour appeler les procédures C en LE_LISP il suffit d'utiliser le nom des procédures définies en C et d'ajouter devant ce nom le caractère "_". Par exemple la procédure TRAITER_EVENEMENTS définie en C sera appelée de la manière suivante :
(_traiter_evenements).

Dans ce fichier la variable globale ROOTWINDOW est définie pour qu'elle soit utilisable en LE_LISP. Cette variable est une manière simple de désigner la fenêtre qui constitue l'écran. En effet X Windows considère l'écran comme la fenêtre père de toutes les fenêtres qui seront créées par la suite.

A2.2 Fichier Xlancement.ll

Des variables globales ont été ajoutées en début de fichier. Ces variables contiennent les polices de caractères utilisées, la hauteur et la largeur des caractères de ces polices, la hauteur et la largeur de l'écran (en pixels et en caractères) ainsi que des noms de fichiers utilisés pour les dessins (pixmap) dans les boîtes à messages. Il est à noter que tous les menus utilisés par la suite sont déclarés en début de ce fichier ce qui facilite donc l'ajout ou la modification d'une sélection d'un menu.

+ SPECIFIE (M)

L'algorithme en pseudo-code de SPECIFIE n'a pas été modifié. Ce sont plutôt les fonctions appelées par la fonction SPECIFIE qui ont été modifiées. Les seules modifications faites sont la modification de toutes les fonctions de manipulation de fenêtres comme par exemple la création de la fenêtre de présentation de SACSO et la suppression de certains messages imprimés à l'écran. De plus tous les appels à la fonction CREE ont été modifiés pour

ajouter les paramètres manquants suite à la modification du nombre de paramètres de la fonction CREE.

+ ENLEVER_LISTE (A)

- argument :

- pré-condition :

- résultat :

0 si la fenêtre courante n'était pas dans la liste FENETRES des fenêtres affichées à l'écran

1 si la fenêtre courante était dans la liste FENETRES

- post-conditions :

Si la fenêtre courante se trouvant dans la variable FENETRE_COURANTE en C se trouvait dans la liste FENETRES des fenêtres affichées à l'écran alors la fenêtre courante ainsi que le record CEYX qui lui est associé sont enlevés de la liste FENETRES, la structure intermédiaire est mise à jour et 1 est renvoyé, sinon 0 est renvoyé.

- note :

La liste des fenêtres affichées à l'écran FENETRES (variable globale) est organisée comme suit : les éléments de numéro pairs (à partir de zéro) sont les instances des fenêtres X Windows obtenues par la fonction CREER_FENETRE et les éléments de numéro impairs sont des records CEYX qui contiennent une description plus précise de la fenêtre X Windows (voir fichier Xinter.ll). Ces records sont d'une importance capitale car ils font le lien entre la structure intermédiaire utilisée pour l'affichage à l'écran et les fenêtres X Windows. Dès lors lorsqu'une fenêtre est tuée ou va être tuée par une instruction C, il s'agit de mettre à jour la liste des fenêtres affichées ainsi que la structure intermédiaire de manière a mettre à jour correctement les structures partagées en commun entre C et LE_LISP.

- algorithme abstrait :

```
début
        {obtenir l'instance de la fenêtre à enlever de la liste}
        fenêtre-id = _retourner_fen_cour

        {chercher si cette fenêtre existe dans la liste et obtenir le record CEYX qui
         correspond à cette fenêtre dans la liste FENETRES}
        fenêtre = chercher_fenêtre (fenêtre-id)

        si fenêtre pas nil
                alors
                        mettre à jour la structure intermédiaire c'est-à-dire enlever
                        dans la s-int les pointeurs vers la fenêtre que l'on va détruire
```

enlever la fenêtre courante et le record de la liste FENETRES

renvoyer 1

sinon

renvoyer 0

fsi

fin

+ ENLEVER_LISTE_FENETRE (A)

- argument :

FENETRE_ID : instance de la fenêtre à enlever de la liste de fenêtres FENETRES

- pré-conditions :

FENETRE_ID doit être l'instance d'une fenêtre obtenue soit par CREER_FENETRE soit par XCreateWindow

- résultat :

- post-conditions :

L'instance de la fenêtre FENETRE_ID ainsi que le record CEYX qui lui est associé sont enlevés de la liste des fenêtres affichées FENETRES.

+ EFFACE_ECRAN (M)

- argument :

- pré-condition :

- résultat :

- post-conditions :

Toutes les fenêtres qui se trouvent dans la liste de fenêtres FENETRES sont détruites et effacées de l'écran. La liste de fenêtres FENETRES est mise à jour, ce qui signifie donc que la liste FENETRES est vide après l'éxécution de EFFACER_ECRAN. Cette fonction utilise la fonction EFFACER_FENETRES qui est définie ci-après.

+ EFFACER_FENETRES (A)

- argument :

LISTE_FENETRES : liste des fenêtres à détruire et donc à effacer à l'écran

- pré-condition :

LISTE_FENETRES doit avoir la même structure que la liste des fenêtres affichées FENETRES (cfr ENLEVER_LISTE)

- résultat :

- post-condition :

Toutes les fenêtres qui se trouvent dans la liste de fenêtres LISTE_FENETRES sont détruites et par conséquent effacées de l'écran.

+ EFFACER_FEN_ERREURS (A)

- argument :

- pré-condition :

- résultat :

- post-conditions :

Toutes les fenêtres qui se trouvent dans la liste (variable globale) FENETRES_DETRUIRE sont détruites et effacées à l'écran, ainsi que la fenêtre d'erreur si il y en a une. La liste de fenêtres FENETRES_DETRUIRE est remplie dans la fonction ATTEND (fichier Xinter-fen.ll).

A2.3 Fichier Xinter-fen.ll

C'est ce fichier qui a subi le plus de modifications. Nous avons essayé de ne pas modifier les paramètres des fonctions de manière à ne pas devoir parcourir tous les fichiers pour modifier les appels des fonctions. Cela n'a toutefois pas été possible pour certaines fonctions.

+ CREE (M)

- arguments :

TITRE : chaîne de caractères qui représente le titre qui sera placé dans la barre de titre de la fenêtre qui sera créée
XPOS : position en x (en pixels) où se trouvera la fenêtre
YPOS : position en y (en pixels) où se trouvera la fenêtre
LARGEUR : largeur en caractères de la fenêtre
HAUTEUR : hauteur en caractères de la fenêtre
NOM : chaîne de caractères qui identifie la fenêtre au niveau interne de la gestion des fenêtres
TYPE_BARRE : paramètre qui sert à dire si on veut attacher à la fenêtre une barre de titre
TYPE_ED : paramètre qui sert à dire si on veut attacher à la fenêtre un éditeur radio
TYPE_FEN : paramètre qui sert à dire si on veut attacher à la fenêtre une fenêtre text
TYPE_MENU : instance du menu que l'on veut attacher à la fenêtre

- pré-conditions :

POSX, POSY ≥ 0

LARGEUR < largeur (ecran)

HAUTEUR < hauteur (ecran)

- résultat :

- post-conditions :

Une fenêtre conforme aux paramètres en entrée est créée.

Si TYPE_BARRE vaut :

* avec_tuer : la fenêtre sera créée avec une barre de titre qui permettra de tuer la fenêtre dans la boîte à sélection droite avec le label T* sans_tuer : la fenêtre sera créée sans boîte à sélection droite et il ne sera donc pas possible de tuer la fenêtre en cliquant sur la boîte à sélection droite.

Pour toute autre valeur la fenêtre sera créée sans barre de titre.

Si TYPE_ED vaut :

* avec_radio : la fenêtre sera créée avec un editeur radio situé juste en dessous de la barre de titre, même si il n'y a pas de barre de titre.

Pour toute autre valeur la fenêtre sera créée sans editeur radio.

Si TYPE_FEN vaut :

* text : la fenêtre sera créée avec une fenêtre text à l'intérieur pour pouvoir afficher du texte à l'intérieur.

Pour toute autre valeur la fenêtre sera créée sans fenêtre text.

Le menu identifié par TYPE_MENU sera attaché à la fenêtre.

+ AFFICHE (M)

- argument :

FENETRE : instance du record CEYX de la fenêtre que l'on veut afficher à l'écran

- pré-condition :

- résultat :

- post-condition :

La fenêtre décrite par l'instance FENETRE est affichée à l'écran. Il s'agit d'afficher tous les composants de la fenêtre comme par exemple la barre de titre, l'editeur radio etc..

+ ECRIRE (M)

- arguments :

FENETRE : instance CEYX de la fenêtre dans laquelle on veut écrire la chaîne de caractères

CHAINE : chaîne de caractères que l'on veut écrire dans la fenêtre

X : position en x (en caractères) où on veut écrire la chaîne

Y : position en y (en caractères) où on veut écrire la chaîne

- pré-condition :
  X, Y ≥ 0

- résultat :

- post-conditions :
  La chaîne de caractères chaîne est affichée dans la fenêtre FENETRE. Comme il n'est pas possible d'écrire une chaîne de caractères à une position donnée dans une fenêtre text, un artifice a été utilisé. Chaque fois que Y vaut soit 0, soit la valeur du Y lors de l'appel précédent à la fonction ECRIRE (mémorisée dans la variable globale Y_ANC), la chaîne de caractères est affichée à la suite de ce qui figure sur la ligne sans passer à la ligne. Dans le cas contraire on passe à la ligne. Dans le casoù on passe à la ligne on vérifie si X est supérieur à 0. Si X est supérieur à 0 on affiche X blancs en début de ligne puis la chaîne de caractères chaîne. Sinon la chaîne de caractères chaîne est affichée en début de ligne.

- algorithme abstrait :

```
début
        si y = 0 OU y = y_anc
                alors
                        on écrit la chaîne chaîne (_ECRIRE_TEXTE_LIGNE)
                sinon
                        on passe à la ligne
                        si x > 0
                                alors on écrit x blancs
                        fsi
                        on écrit la chaîne chaîne (_ECRIRE_TEXTE_LIGNE)
        fsi
        y_anc = y
fin
```

+ EFFACE (M)

- argument :
  FENETRE : instance CEYX de la fenêtre que l'on veut effacer et détruire à l'écran

- pré-condition :
  La fenêtre FENETRE doit être affichée à l'écran

- résultat :

- post-conditions :
  La fenêtre FENETRE est effacée de l'écran. La fonction utilisée pour effacer la fenêtre est la fonction C _TUER_FENETRE qui détruit physiquement la fenêtre. La

fenêtre FENETRE ainsi que l'instance X Windows de la fenêtre correspondante sont enlevées de la liste de fenêtres FENETRES et la structure intermédiare est mise à jour pour refléter qu'une fenêtre a été détruite. Tous les liens vers cette fenêtre dans la structure intermédiaire sont détruits. La fonction EFFACE et la fonction DETRUIRE sont identiques car X Windows procède toujours à une destruction physique de la fenêtre. La structure intermédiare est donc mise à jour de la même manière que dans la fonction DETRUIRE de SACSO.

+ ID_XW (A)

- argument :

  FENETRE : instance CEYX de la fenêtre dont on veut obtenir l'instance X Windows c'est-à-dire le pointeur vers la fenêtre réelle, physique X Windows

- pré-condition :

- résultat :

  Instance de la fenêtre X Windows

- post-condition :

  L'instance de la fenêtre X Windows est renvoyée c'est-à-dire le champ ID_FEN du record fenetre-reelle de fenetre (voir fichier Xinter.ll).

+ ID_TEXT (A)

- argument :

  FENETRE : instance CEYX de la fenêtre dont on veut obtenir l'instance X Windows de la fenêtre text qu'elle contient

- pré-condition :

- résultat :

  Instance de la fenêtre text X Windows

- post-condition :

  L'instance de la fenêtre text X Windows est renvoyée c'est-à-dire le champ FEN_TEXT du record fenetre-reelle du record fenêtre (voir fichier Xinter.ll).

+ ID_XTXTEDIT (A)

- argument :

  FENETRE : instance CEYX de la fenêtre dont on veut obtenir l'instance X Windows de l'éditeur de texte qu'elle contient

- pré-condition :

- résultat :

Instance de l'éditeur de texte X Windows c'est-à-dire le champ ID_TXTEDIT du record fenetre-reelle du record fenetre (voir fichier Xinter.ll)

- post-condition :

L'instance de l'éditeur de texte X Windows est renvoyée.

+ CREER_RADIO_BOUT (A)

- arguments :

FENETRE : instance CEYX de la fenêtre dans laquelle on veut créer un éditeur radio

CHAMP1... CHAMP4 : chaînes de caractères qui constitueront les labels placés à côté des boutons de l'éditeur radio

NB_COLS : nombre de colonnes de l'éditeur radio

POLICE_TITRE : police utilisée pour la barre de titre de la fenêtre dans laquelle sera placé l'éditeur radio

POLICE_RADIO : police à utiliser pour l'éditeur radio

- pré-conditions :

NB_COLS > 0

- résultat :

- post-conditions :

Un éditeur radio est créé dans l'instance de la fenêtre FENETRE et tous les champs de l'instance FENETRE sont mis à jour pour refléter la présence de l'éditeur radio.

- algorithme abstrait :

début

{obtenir l'instance X Windows de la fenêtre}
fenetre-id = id_XW (fenetre)

{création de l'éditeur radio (_CREER_RADIO_BOUT)}
radio_bouton = _creer_radio_bout (fenetre_id, ... , police_radio)

{obtenir la hauteur de l'éditeur radio (_OBTENIR _HAUT_RADIO_BOUT)}
yposed = _obtenir_haut_radio_bout (champ1, ..., police_radio)

calcul pour connaître la valeur de la position en y en dessous de l'éditeur radio

mise à jour des champs du record CEYX qui décrit la fenêtre

si existe fenêtre text
alors

{agrandir la fenêtre qui contient la fenêtre text puisque la position de la fenêtre text va changer}
(_CHANGER_TAILLE_FENETRE)

{changer la position de la fenêtre text pour qu'elle se trouve en dessous de l'éditeur radio}
(_CHANGER_POSITION_FENETRE)

fsi

fin

+ DESAFFICHE (A)

- argument :

FENETRE : instance CEYX de la fenêtre que l'on va effacer à l'écran sans la tuer c'est-à-dire qu'il sera toujours possible de la réafficher par la suite

- pré-condition :

- résultat :

- post-condition :

La fenêtre décrite par l'instance FENETRE est effacée de l'écran et pourra être réaffichée par la suite puisqu'elle n'est pas détruite physiquement.

+ RAFFICHE (A)

- argument :

FENETRE : instance CEYX de la fenêtre que l'on veut réafficher à l'écran

- pré-condition :

L'instance CEYX FENETRE a été éffacée précédemment de l'écran grâce à la fonction DESAFFICHE

- résultat :

- post-condition :

La fenêtre décrite par l'instance FENETRE est réaffichée à l'écran.

+ CHERCHER_FENETRE_XW (A)

Renvoie l'instance X Windows de l'instance de la fenêtre FENETRE.

+ ATTEND (A)

L'instance CEYX de la fenêtre passée en arguments à la fonction ATTEND est mise dans la variable globale FENETRES_DETRUIRE. Toutes les fenêtres de la liste

FENETRES_DETRUIRE seront effacées par la suite dans la fonction SPECIFIE en éxécutant EFFACER_FEN_ERREURS lorsque l'utilisateur éxécutera une autre commande.

A2.4 Fichier Xsacso.ll

Seule la fonction charger et la fonction visu ont été modifiées dans le fichier Xsacso.ll. Tous les appels à la fonction CREE ont été remplacés pour avoir le nombre correct de paramètres. Pour les fonctions dupliquer, biblio, detruire, charger un test de l'indicateur indicateur_commande a été ajouté avant d'appeler l'éditeur par INIT_EDIT. Si cet indicateur est positionné cela signifie que l'utilisateur a déjà pointé sur un objet avec le système de pointage et qu'il veut donc que la commande qu'il va éxécuter s'applique à cet objet. Si l'indicateur est positionné, il n'est donc pas nécessaire d'appeler l'éditeur pour obtenir le nom de l'objet auquel l'utilisateur veut appliquer la commande.
La fonction {Fenetre}:visu n'est plus nécessaire, d'où son corps vide pour éviter de devoir supprimer tous les appels de cette fonction dans tous les fichiers.

+ VISU (M)

- arguments :
  NOEUD : noeud que l'on veut visualiser dans la fenêtre
  TYPE_AIDE : chaîne de caractères qui constituera le titre de la fenêtre qui sera créée pour afficher le noeud NOEUD

- pré-condition :

- résultat :

- post-condition :
  Une fenêtre, avec le titre TYPE_AIDE et ayant pour contenu le noeud NOEUD est créée et affichée à l'écran.

+ CHARGER (M)

La structure de la fonction CHARGER n'a pas été modifiée. Les modifications apportées sont les suivantes :

- au lieu de tuer les fenêtres du niveau 1 en passant au niveau 2 elles sont effacées simplement de l'écran (fonction DESAFFICHE) pour pouvoir les réafficher par la suite. L'éditeur de texte par contre est détruit,
- création d'une fenêtre pour indiquer que le chargement est en cours,
- à la sortie de charger il y a création de l'éditeur, réaffichage des fenêtres. Si la spécification est nouvelle alors il y a mise à jour de la variable existant qui contient la liste des spécifications disponibles et réaffichage des spécifications disponibles dans la fenêtre "Spécifications disponibles".

- algorithme abstrait de charger (nom-specif):
début
  {Changement du niveau de commande}
  niveau = "sacso:chargee"

si nom-specif = vide
alors
si indicateur_commande positionné
sinon
lecture du nom de la spécif à charger
fsi
fsi
on efface les fenêtres du niveau 1 et on tue l'éditeur
création de la fenêtre pour indiquer que le chargement est en cours
.
. voir ancienne documentation
.
retour au niveau précédent
on restaure les fenêtres c'est-à-dire qu'on réaffiche les fenêtres qui avaient été effacées grâce à la fonction DESAFFICHE et qui se trouvaient dans la liste FENETRES_RESTAURER

si (membre (nom-specif) existant)
alors
sinon
on met à jour la liste existant qui contient la liste des spécifications disponibles
on réaffiche les spécifications disponibles dans la fenêtre des spécifications disponibles
fsi
fin

A2.5 Fichier Xballade.ll

Le fichier Xballade.ll comprend les fonctions de scrolling dans les fenêtres ainsi que la procédure responsable du pointage. Un certain nombre de fonctions inutiles ont été supprimées. Dans toutes ces fonctions les appels à la fonction NOUV-SI, PREMIER et CHG-ETAT-COURANT ont été supprimés car ces fonctions sont responsables du calcul de l'entité à remettre en reverse vidéo lorsque l'on scrolle dans la fenêtre. Comme dans XSACSO il n'y a plus de reverse vidéo les appels à ces fonctions ont été supprimés.

+ CHG-FENETRE (M)

- arguments :
X_POS : position en x (en pixels) où l'utilisateur a pointé dans la fenêtre
Y_POS : position en y (en pixels) où l'utilisateur a pointé dans la fenêtre

- pré-condition :
X_POS, Y_POS > 0

- résultat :

- post-conditions :

La fonction CHG-FENETRE positionne la variable globale si-cour à la structure pointée par l'utilisateur. Avec les positions reçues en entrée cette fonction va aller voir dans la structure intermédiaire associée à la fenêtre, quelle est la partie de la structure intermédiaire sélectionnée. La fonction CHG-FENETRE teste d'abord l'indicateur chg_fen_deja_execute pour s'assurer qu'il n'y a pas déjà un pointage en cours. En effet, on n'autorise l'utilisateur qu'à effectuer un seul pointage à la fois.

- algorithme abstrait :

```
début
        si chg_fen_deja_execute est sans valeur (pas de pointage en cours)
         alors
                {obtenir l'instance X Windows de la fenêtre dans laquelle l'utilisateur a
                pointé (_RETOURNER_FEN_COUR)}
                fenetre-id = _retourner_fen_cour

                dépositionnement de l'indicateur de commande

                {chercher la fenêtre CEYX à laquelle correspond cette instance
                (CHERCHER_FENETRE)}
                fenetre = chercher_fenetre (fenetre-id)

                si fenetre pas vide
                        alors
                                transformations des coordonnées reçues en pixels en
                                caractères, on obtient x_ligne, y_ligne

                                {détermination de la partie de la structure intermédiaire à
                                laquelle correspondent ces caractères et affectation à la
                                variable si-cour}
                                si-cour = designe (x_ligne, y_ligne, fenetre)

                                si si-cour pas nulle et champ sons <> type arbre et sons est
                                une  chaîne de caractères
                                alors
                                        {lecture du pixmap que l'on mettra dans la boîte à
                                        messages}
                                        pixmap = _lire_pixmap FICHIER_POINT_-
                                                PIXMAP

                                        {désactiver tout l'éditeur}
                                        (_DEACTIVER_TXTEDIT)
```

{désactiver l'éditeur à boutons de l'éditeur de texte} (_DEACT_ED_BOUTON)

donner une valeur à l'indicateur chg_fen_deja_execute

{création d'une boîte à messages pour recevoir les commandes de l'utilisateur (zoom, modification, + de détail,...)}
(_CREER_boîte_MESSAGE)

réactiver l'éditeur de texte

réactiver l'éditeur à boutons

enlever la valeur de chg_fen_deja_execute

fsi

fsi

fsi

fin

- note :

A la fin du fichier figure un certain nombre de redéfinitions de noms de fonctions grâce à la fonction synonym LE_LISP. Ceci est nécessaire car ces fonctions sont appelées directement depuis C qui ne va pas chercher les fonctions dans les packages mais les cherche dans le package #:user. Ces fonctions doivent donc avoir leur nom défini dans le package #:user.

A2.6 Fichier Xchargee.ll

Modification des appels à la fonction CREE pour prendre en compte ses nouveaux paramètres (cfr. Xinter-fen.ll). De plus nous avons ajouté un test d'indicateur positionné si l'utilisateur a déjà sélectionné un objet au moyen du pointage. On teste pour cela la variable indicateur_commande qui a une valeur lorsque l'utilisateur a cliqué dans le champ commande de la boîte à messages. Dans ce cas il n'est pas nécessaire d'appeler init-edit (active l'éditeur) pour saisir le nom de l'objet sur lequel va porter la commande puisque ce nom se trouve dans la variable si-cour. Ce test a été ajouté dans les fonctions DUPLIQUER, DETRUIRE.

A2.7 Fichier Xconstruire.ll

Modification des appels à la fonction CREE pour prendre en compte ses nouveaux paramètres (cfr. Xinter-fen.ll).

A2.8 Fichier Xdétruire.ll

Modification des appels à la fonction CREE pour prendre en compte ses nouveaux paramètres (cfr. Xinter-fen.ll).

A2.9 Fichier Xerreur.ll

Modification des appels à la fonction CREE pour prendre en compte ses nouveaux paramètres (cfr. Xinter-fen.ll). La fonction ATTEND devient inutile, son corps est vide mais sa déclaration a été conservée pour éviter de devoir modifier tous les appels à cette fonction dans tous les fichiers. La fonction RAZ n'efface plus la fenêtre d'erreur car celle-ci est effacée lorsque l'utilisateur change de commande.

A2.10 Fichier Xinterprete.ll

Modification des appels à la fonction CREE pour prendre en compte ses nouveaux paramètres (cfr. Xinter-fen.ll). Seuls tous les appels à la fonction CREE ont été changés dans ce fichier pour ajouter les paramètres manquants suite à la modification du nombre de paramètres de la fonction CREE.

A2.11 Fichier Xinter.ll

Le fichier Xinter.ll contient la description du record Fenetre utilisé comme paramètre dans la plupart des procédures. Sa définition a été modifiée par adjonction de champs nécessaires à la composante Fenetre-Reelle du record Fenetre. Un fenêtre peut être composée d'une barre de titre et/ou d'un éditeur à boutons et/ou d'un éditeur radio et/ou d'une fenêtre text et/ou d'un éditeur de texte. Le record Fenetre se présente comme suit :

(deftrecord **Fenetre-Reelle**

(titre ()) chaîne de caractères constituant le titre de la fenêtre
(xpos 0) position en x à laquelle on peut écrire dans la fenêtre
(ypos 0) postion en y à laquelle on peut écrire dans la fenêtre
(largeur 0) largeur en caractères de la fenêtre
(hauteur 0) hauteur en caractères de la fenêtre
(larg_pix 0) largeur en pixels de la fenêtre
(haut_pix 0) hauteur en pixels de la fenêtre
(id_fen) instance X Windows de la fenêtre c'est-à-dire pointeur vers la fenêtre X Windows
(barre) instance de la barre de titre attachée à la fenêtre
(ed_bout) instance de l'éditeur à boutons
(radio_bout) instance de l'éditeur radio
(fen_text) instance de la fenêtre text
(id_txtedit) instance de l'éditeur de texte

)

+ AFFICHE (M)

La partie qui se situe après le commentaire Scroll si nécessaire dans la fonction a été supprimé du fait que le pointage à l'écran ne nécessite plus de faire du scrolling dans la fenêtre.

D'autre part les fenêtres créées par la fonction affiche lorsque l'on demande d'afficher dans une fenêtre qui n'existe pas, n'ont plus leur origine placée à la position du curseur.

Cas 1 : les fenêtres ont leur coin gauche supérieur positioné comme suit :
la première fenêtre à la position (0, 0)
la deuxième fenêtre en (0 + X_DEPL, 0 + Y_DEPL) sauf si la largeur de la fenêtre est trop grande pour que son coin supérieur gauche puisse être placé à cette position. Si c'est le cas son coin supérieur gauche est placé en (0, 0 + Y_DEPL) et ainsi de suite pour les autres fenêtres jusqu'à ce que la somme des Y_DEPL soit supérieure à sept fois la hauteur du titre.

Cas 2 : les fenêtres ont leur coin droit supérieur positioné comme suit :
la première en (largeur(ecran), 0)
la deuxième en (largeur(ecran) - X_DEPL, 0 + Y_DEPL) sauf si la largeur de la fenêtre est trop grande pour avoir son coin supérieur droit placé à cette position. Si c'est le cas son coin supérieur droit est placé en (0, 0 + Y_DEPL) et ainsi de suite jusqu'à ce que la somme des Y_DEPL soit supérieure à sept fois la hauteur du titre. Dans ce cas les fenêtres sont positionnées comme dans le cas 1.

+ REAFFICHE (M)

La fonction REAFFICHE n'a été modifiée que pour le réaffichage des fenêtres. En effet comme avec X Windows il n'est pas possible d'écrire une ligne dans une fenêtre text à une position en x et en y donnée, il n'est pas possible de réafficher uniquement la partie de la fenêtre modifiée mais il faut réafficher tout le contenu de la fenêtre si une partie de cette fenêtre a été modifiée. C'est pour cette raison qu'une partie de la fin de la procédure REAFFICHER a été supprimée. On ne calcule donc plus la partie de la fenêtre ou des fenêtres à réafficher mais on teste simplement si la partie modifiée se trouve affichée dans une fenêtre à l'écran. Si c'est le cas on réaffiche la nouvelle structure intermédiaire correspondante à la fenêtre cette structure intermédiaire ayant été mise à jour, sinon on ne réaffiche rien.

+ EFFACE (M)

Avec X Windows, effacer la s-int dans une fenêtre revient à effacer le contenu de la fenêtre text qui contient le texte de la structure intermédiaire. En effet le texte de la structure intermédiaire est toujours affiché à l'écran dans une fenêtre text.

+ DETRUIT (M)

La fonction DETRUIT devient un appel à la procédure EFFACE.

+ {specif}:DESIGNE (M)

La fonction DESIGNE retourne la structure intermédiaire correspondant à la partie de texte affichée à la position (X, Y) dans la fenêtre Ceyx identifiée par FENETRE. Cette fonction est utilisée pour le pointage. Lorsque l'utilisateur pointe sur un objet, le système de pointage récupère la position en x et en y de la souris (fonction CHG-FENETRE), CHG-FENETRE appelle DESIGNE pour obtenir la structure intermédiaire correspondant à cette position.

A2.12 Fichier XWfedit.ll

Ce fichier remplace le fichier FEDIT.LL duquel toutes les fonctions ont été modifiées ou supprimées. Ce fichier regroupe une partie des fonctions lisp ayant rapport avec la gestion de l'éditeur de texte (cfr. XWlancement.ll pour l'autre partie).

Note

De façon générale l'éditeur n'est sensible que lorsque le programme exécute la fonction TRAITER_EVENEMENTS() et "insensible" dans les autres cas. Il est cependant un cas particulier où l'éditeur est insensible pendant l'exécution de la fonction TRAITER_EVENEMENTS(). En effet lorsque l'éditeur est sensible et activé, il a la particularité de récupèrer tous les événements quels qu'ils soient. Ce qui est très gênant lorsque certains de ces événements doivent être traités par d'autres "éditeurs" comme une boîte à messages. C'est pourquoi, pendant l'exécution de TRAITER_EVENEMENTS (), l'éditeur de texte est rendu "insensible avant de traiter les événements de la boîte à messages.

+ EDIT (M)

La fonction EDIT retourne le texte qui a été édité dans la fenêtre d'édition.

- argument :
  FENETRE fenêtre d'édition

- pré-conditions :
  La fenêtre d'édition est déjà créée
  L'éditeur de texte est insensible

- résultat :
  Le texte édité

- post-conditions :
  Le texte renvoyé est une liste de chaines de caractères résultat de la transformation du texte édité en une liste à interpréter. En sortie de l'édition, l'éditeur de texte est insensible.

- note :

Si le choix de l'utilisateur porte sur une "commande menu" on quitte momentanément l'éditeur par exécution de la commande, sinon la fonction renvoie le contenu du tampon d'édition.

La fonction TRAITER_EVENEMENTS() renvoit une valeur qui identifie le choix de l'utilisateur (commande menu, commande éditeur, autre). Elle renvoit une chaîne de caractères vide lorsque l'utilisateur passe par l'éditeur de texte sinon elle renvoit une chaîne de caractères dont la valeur identifie la commande à exécuter.

- algorithme abstrait :

début

initialisation du buffer d'édition (...)

affichage du buffer d'édition

choix := TRAITER_EVENEMENTS () {Que veut l'utilisateur ?}
désactivation de l'éditeur de texte (insensible, inactif)

tant que (choix est différent d'une édition de texte) faire
si choix = annuler
alors annuler commande qui a appelé edit
sinon
sauvegarde de l'édition en cours sur pile-edit

si (vérification syntaxique et lexicale de la commande)=OK
alors traiter la commande menu (on quitte l'éditeur)
sinon message d'exception
fsi

restauration de l'édition sauvée sur pile-edit
activation de l'éditeur de texte (sensible, actif)

choix := TRAITER_EVENEMENTS () {Que veut l'utilisateur ?}
désactivation de l'éditeur de texte
fsi
ffaire

récupération du texte édité

{la transformation du texte édité en une liste, est nécessaire pour les fonctions qui appellent edit et qui traite le résultat}
liste = transformation en une liste du texte édité
retourner (liste)
fin

+ EDIT:FIN-EDIT (M)

- argument :

- pré-condition :
L'éditeur de texte est sensible

- résultat :

- post-conditions :
Cette fonction éteind le curseur de la fenêtre d'édition.
L'éditeur de texte est insensible.

+ EDIT:REFAIRE-AFF (M)

- argument :

- pré-condition :
  L'éditeur de texte est insensible.

- résultat :

- post-conditions :
  Elle réaffiche la fenêtre d'édition.
  L'éditeur de texte est insensible.

- définition de synonyme
  SYN_EDIT_HELP() synonyme de la fonction LE_LISP EDIT:HELP()

A2.13 Fichier XWlancement.ll

Ce fichier reprend du fichier LANCEMENT.LL l'ensemble des fonctions ayant rapport avec l'édition de texte. En dehors des fonctions COMMANDE-SAISIE() et FIN-EDIT() les autres fonctions du fichier LANCEMENT.LL ont été modifiées.
Dans le système, la saisie de texte se fait toujours dans la même fenêtre F-EDIT.
Pendant une édition de texte, l'utilisateur peut faire d'autres actions, comme sélectionner avec la souris une autre fenêtre (cfr. la fonction BALLADE) ou lancer l'exécution d'une commande à l'aide du menu. Auquel cas, il est nécessaire de pouvoir suspendre l'édition en cours et de pouvoir y revenir avec la fenêtre d'édition qui correspond à l'activité suspendue afin de la terminer. C'est le rôle de la variable PILE-EDIT sur laquelle on "empile" et "dépile" afin de respectivement sauver et restaurer la fenêtre d'édition F-EDIT, en cours lors de l'appel à la fonction INIT-EDIT.

DEFINITION DE VARIABLES GLOBALES

POLICE_MESSAGE : définit la police de caractères utilisée pour écrire un texte dans la zone message de l'éditeur
POLICE_TXTEDIT : définit la police de caractères utilisée pour écrire un texte dans la zone d'édition de l'éditeur
POLICE_AIDE : définit la police de caractères utilisée pour écrire un texte dans la zone aide de l'éditeur. Celle-ci correspond au "mini-aide" du système SACSO avant modifications
HAUTEUR_AIDE : définit la hauteur en pixel d'un caractère de la police POLICE_AIDE
HAUTEUR_MESSAGE : définit la hauteur en pixel d'un caractère de la police POLICE_MESSAGE
HAUTEUR_TXTEDIT : définit la hauteur en pixel d'un caractère de la police POLICE_TXTEDIT

LARGEUR_MESSAGE : définit la largeur en pixel d'un caractère de la police POLICE_MESSAGE
LARGEUR_TXTEDIT : définit la largeur en pixel d'un caractère de la police POLICE_TXTEDIT
FICHIER_VALID_PIXMAP : définit le chemin "unix" qui permet d'accéder au fichier bitmap définissant la figure (pixmap). Ce pixmap est utilisé dans les boîtes à messages

+ INIT-EDIT (M)

Cette fonction initialise la fenêtre d'édition pour une nouvelle édition.

- argument :

  TITRE : titre de la fenêtre

- pré-conditions :

  L'éditeur de texte, s'il existe est insensible
  La pile PILE-EDIT est définie

- résultat :

- post-conditions :

  L'éditeur de texte est insensible.
  Si la fenêtre d'édition n'existe pas déjà, elle est créée et affichée. Le titre TITRE est affiché dans la zone message de l'éditeur. Si la fenêtre d'édition existe déjà, les zones d'édition et de messages sont sauvegardées dans la pile PILE-EDIT.

- algorithme abstrait :

```
début
    si la fenêtre d'édition existe
        alors
            ajouter sur pile-edit le titre, la s-int, le buffer
            afficher le titre TITRE de la nouvelle fenêtre d'édition
        sinon
            création de la fenêtre d'édition
            affichage de la fenêtre d'édition
    fsi
fin
```

+ RESTAURE-EDIT (M)

Cette fonction est utilisée pour restaurer la fenêtre d'édition à partir de la pile d'édition PILE-EDIT.

- argument :

- pré-conditions :

L'éditeur de texte est insensible

La pile PILE-EDIT est définie

- résultat :

- post-conditions :

L'éditeur de texte est sensible.

La fenêtre d'édition sommet de la pile PILE-EDIT (zones d'édition et de messages), est restaurée et est affichée. Après restauration, l'éditeur de texte est rendu sensible aux événements pour reprendre l'activité d'édition correspondant à la fenêtre restaurée.

La fenêtre restaurée est retirée de la pile PILE-EDIT.

- algorithme abstrait :

```
début
        si pile-edit=vide
                alors effacer la fenêtre d'édition
                sinon
                        si structure intermédiaire est associée à la fenêtre d'édition
                                alors défaire la structure intermédiaire de la fenêtre
                                d'édition
                        enlever la fenêtre de la pile-edit
                        afficher la fenêtre
                        activer l'éditeur à boutons associé (sauf le bouton aide)
                        activer l'éditeur de texte associé
        fsi
fin
```

+ TRAITER-COMMANDE (M)

Cette fonction s'occupe du traitement d'une commande de l'utilisateur, intoduite au moyen du clavier ou de la souris.

- argument :

- pré-condition :

L'éditeur de texte est insensible

- résultat :

- post-conditions :

Cette fonction effectue la saisie et l'exécution de la commande.

Les zones d'édition, de messages, et d'aide sont réinitialisées pour le traitement d'une autre commande.

L'éditeur de texte est insensible.

+ CHGT-POLICE-EDIT (A)

- argument :

  POLICE : la police de caractères

- pré-condition :

  La police de caractères POLICE existe

- résultat :

- post-conditions :

  Lorsqu'il est impossible d'utiliser la police de caractères définie par défaut pour la zone d'édition, cette fonction appelée depuis les primitives C, permet de changer les variables globales LE_LISP POLICE_TXTEDIT, LARGEUR_TXTEDIT, HAUTEUR_TXTEDIT.

- définition de synonyme :

  SYN_CHGT_POLICE_EDIT() synonyme de la fonction LE_LISP CHGT-POLICE-EDIT()

A2.14 Fichier XWinter-fen.ll

Les fonctions décrites dans ce fichier complètent le fichier INTER-FEN.LL.

+ CREER_EDITFEN (A)

- arguments :

  TITRE : titre de la fenêtre
  XPOS : position x en pixel
  YPOS : position y en pixel
  LARGEUR : largeur en terme du nombre de colonnes
  HAUTEUR : hauteur en terme du nombre de lignes
  NOM : nom de la fenêtre
  TYPE_BARRE : paramètre désignant le type de barre de titre
  TYPE_ED : paramètre désignant le type d'éditeur à bouton
  TYPE_FEN : paramètre désignant le type de fenêtre texte
  TYPE_MENU : paramètre désignant le type de menu

- pré-condition :

- résultat :

  FENETRE : fenêtre Ceyx

- post-conditions :

FENETRE identifie une fenêtre d'édition Ceyx créée.

Cette fonction crée la fenêtre d'édition avec le type de barre de titre TYPE_BARRE, le type de la zone de message TYPE_FEN (fenêtre texte), le type de la zone d'aide TYPE_ED (éditeur à boutons correspondant au "mini-aide" du système SACSO avant modifications), le titre TITRE, la largeur LARGEUR, la hauteur HAUTEUR donnés en paramètre. Cette fenêtre dispose également d'une zone d'édition.

Les polices de caractères utilisées pour les différentes zones de l'éditeur, sont définies par défaut.

- algorithme abstrait :

```
début
        obtenir une instance du record Ceyx Fenetre
        positionner la fenêtre pour ne pas dépasser l'écran (XPOS, YPOS)
        si TYPE_BARRE="avec tuer" ou "sans tuer"
                alors {créer une barre de titre suivant le paramètre}
                        (_CREER_BARRE_TITRE)
        fsi
        si TYPE_FEN="text"
                alors {créer une fenêtre texte suivant le paramètre}
                        (_CREER_FENETRE_TEXT)
        fsi
        ajuster la taille de la fenêtre
        si TYPE_ED="avec_radio"
                alors {créer un éditeur radio avec les boutons annuler commande,
                        envoyer commande, description paramètre, aide}
                        (_CREER_ED_BOUT)
        fsi
        si TYPE_MENU
                alors {attacher à la fenêtre le menu de type TYPE_MENU}
                        (_ATTACHER_MENU_FEN)
        fsi
        ajuster la taille de la fenêtre
        {créer un éditeur de texte}
        (_CREER_TXTEDIT)
fin
```

+ AFFICHE_EDITFEN (A)

- argument :

FENETRE fenêtre Ceyx

- pré-condition :

FENETRE_ID identifie une fenêtre existante

- résultat :

-post-condition :

La fenêtre Ceyx identifiée par FENETRE_ID est affichée à l'écran.

+ CREER_ED_BOUT (A)

- arguments :

FENETRE : fenêtre Ceyx
CHAMP1,CHAMP2,...,CHAMP4 : boutons de l'éditeur
NB_CHAMP : nombre de boutons actifs
NB_COLS : nombre de colonnes
POLICE_TITRE : police du titre
POLICE_ED : police de caractères utilisée pour le texte des boutons

- pré-condition :

FENETRE identifie une fenêtre Ceyx existante

- résultat :

ED_BOUT_ID : instance de l'éditeur à boutons créé

- post-conditions :

Cette fonction crée un éditeur à boutons avec les NB_CHAMP boutons qui sont CHAMP1, CHAMP2, CHAMP3, CHAMP4 rangés en NB_COLS colonnes, dont la police de caractères est POLICE_ED.

Cet éditeur est rattaché à la fenêtre FENETRE. La police de caractères du titre POLICE_TITRE doit être donnée pour pouvoir placer l'éditeur en dessous de la barre de titre. Si nécessaire, la taille et la position des autres "éditeurs" de la fenêtre FENETRE seront modifiées de telle sorte que la fenêtre se présente, de haut vers le bas, avec en premier la barre de titre, puis l'éditeur à boutons, et ensuite la zone de messages.

+ ECR_MESSAGE (A)

- arguments :

FENETRE : fenêtre Ceyx
MESSAGE : texte à ecrire

- pré-condition :

FENETRE identifie une fenêtre Ceyx existante

- résultat :

- post-conditions :

Cette fonction affiche un message MESSAGE dans la zone de messages de la fenêtre d'édition FENETRE.

L'ancien message de cette zone est perdu.

+ POP (M)

- argument :

FENETRE : fenêtre Ceyx

- pré-conditions :

FENETRE identifie une fenêtre d'édition existante

L'éditeur de texte et l'éditeur à boutons de la fenêtre d'édition sont insensibles

- résultat :

- post-conditions :

L'éditeur de texte et l'éditeur à boutons de la fenêtre d'édition sont sensibles.

+ {Fenetre}:affiche-buffer (A)

- argument :

FENETRE_ID : la fenêtre d'édition

- pré-conditions :

FENETRE_ID identifie une fenêtre d'édition existante

L'éditeur de texte est insensible

- résultat :

- post-conditions :

Affiche le contenu du buffer d'édition dans la fenêtre d'édition.

L'éditeur de texte est insensible.

A2.15 Fichier XWrenomme.ll

Seule la fonction VALIDATION() du fichier RENOMME.LL a été modifiée et donc reprise dans ce fichier.

+ VALIDATION

- argument :

- pré-condition :

- résultat :

- post-condition :

Cette fonction permet de valider ou pas l'opération de renommage d'une spécification. Elle propose à l'aide d'une boîte à messages, les choix d'annulation, de validation, ou de non validation.

A2.16 Fichier XWmodifie.ll

Ce fichier reprend certaines fonctions du fichier MODIFIE.LL.

Comme pour le renommage, la validation d'une modification ne se fait plus à l'aide de l'éditeur mais par la création d'une boîte à messages (cfr. XWRENOMME.LL).

Les fonctions **{BLOCK-OP}:MODIFIE** et **{BLOCK-TYPE}:MODIFIE** ont donc été modifiées en conséquence.

D'autres fonctions ont également été redéfinies dans le seul but d'éliminer le message d'erreur "RETOUR A L'EDITEUR PAR ^R" qui dans XSACSO n'a plus de sens .

Etant donné le peu de modifications apportées, il n'est pas nécessaire d'énumérer ces fonctions.

A2.17 Fichier XWchargee.ll

Les modification des changements de titre d'une fenêtre sont remplacés par un appel à ecr_message avec le titre en paramètre. De plus, une boîte à messages est utilisée pour valider le sauvetage d'une spécification dans la fonction QUITTER.

A2.18 Fichier XWaide.ll

Les fonctions du fichier AIDE.LL ont été reprises ici afin de mettre à jour simplement l'aide en ligne. Le synonyme SYN_AIDE_EDITEUR() a été défini pour permettre d'obtenir de l'aide en ligne sur l'éditeur de texte à partir du bouton AIDE-EDITEUR de la zone d'aide de cet éditeur.

- définition de synonyme :

SYN_AIDE_EDITEUR() synonyme de SACSO:AIDE(editeur)

Annexe 3

Cette annexe reprend tous les types de base et constructeurs de types du langage SACSO. Pour chaque type de base et constructeur de types les opérations disponibles sont spécifiées.

A3.1 Les types

3.1.1 Les types de base

BOOL : type BOOLEEN
ENTIER : type ENTIER
F-ELEM : type paramètre formel
F-ELEM1 : type paramètre formel
CHAINE : type des chaînes de caractères
CAR : type d'un caractère

3.1.2 Les constructeurs de types

PC[**F-ELEM, +**] : type PRODUIT CARTESIEN
E[**F-ELEM**] : type ENSEMBLE
S[**F-ELEM**] : type SUITE
U[**F-ELEM, +**] : type UNION
T[**F-ELEM, F-ELEM1**] : type TABLE

A3.2 Opérations possibles sur les types

3.2.1 Opérations associées aux types de base

3.2.1.1 Opérations associées à BOOL

eq-bool : **BOOL, BOOL -> BOOL**
égalité pour les booléens

et : **BOOL, BOOL -> BOOL**
et logique

faux : **-> BOOL**
faux

impl : **BOOL, BOOL -> BOOL**
implication logique

non : **BOOL -> BOOL**
négation logique

or : **BOOL, BOOL -> BOOL**
ou logique

vrai : **-> BOOL**
vrai

3.2.1.2 Opérations associées à ENTIER

eq-entier : **ENTIER, ENTIER -> BOOL**
le premier entier est-il égal au deuxième ?

indef-entier : **-> ENTIER**
indéfini pour les entiers

infeg : **ENTIER, ENTIER -> BOOL**
le premier entier est-il inférieur ou égal au deuxième ?

inférieur : **ENTIER, ENTIER -> BOOL**
le premier entier est-il inférieur au deuxième ?

max : **ENTIER, ENTIER -> ENTIER**
maximum de deux entiers

min : **ENTIER, ENTIER -> ENTIER**
minimum de deux entiers

moins : **ENTIER, ENTIER -> ENTIER**
soustraction de deux entiers

nul : **ENTIER -> BOOL**
un entier est-il égal à zéro ?

plus : **ENTIER, ENTIER -> ENTIER**
addition de deux entiers

suc : **ENTIER -> ENTIER**
incrémentation d'un entier

supérieur : **ENTIER, ENTIER -> BOOL**
le premier entier est-il supérieur au deuxième ?

zéro : **-> ENTIER**
entier zéro

3.2.1.3 Opérations associées à F-ELEM

eq-f : **F-ELEM, F-ELEM -> BOOL**
égalité pour F-ELEM

indef-f : **-> F-ELEM**
indéfini pour F-ELEM

3.2.1.4 Opérations associée à F-ELEM1

eq-f1 : **F-ELEM1, F-ELEM1 -> BOOL**
égalité pour F-ELEM1

3.2.1.5 Opérations associées à CHAINE

concat : **CHAINE, CHAINE -> CHAINE**
concaténation de deux chaînes de caractères

eq-chaîne : **CHAINE, CHAINE -> BOOL**
la première chaîne est-elle égale à la deuxième ?

extrait : **CHAINE, ENTIER, ENTIER -> CHAINE**
extraction d'une sous-chaîne de caractères débutant à une certaine position et se terminant à une certaine position

indef-chaîne : **-> CHAINE**
indéfini pour le type CHAINE

3.2.2 Opérations associées aux constructeurs de types

3.2.2.1 Opérations associées à PC

accès : PC[**F-ELEM, +**], **ENTIER -> F-ELEM1**
accès à la valeur d'un champ d'un produit cartésien
(peut être utilisée plus agréablement par F-ELEM(produit cartésien))

eq-pc : PC[**F-ELEM, +**], PC[**F-ELEM, +**] **-> BOOL**
égalité de deux produits cartésiens

indef-pc : **->** PC[**F-ELEM, +**]
indéfini pour le produit cartésien

modif : PC[**F-ELEM, +**], **F-ELEM1, ENTIER ->** PC[**F-ELEM, +**]
modification de la valeur d'un champ d'un produit cartésien

3.2.2.2 Opérations associées à E

adj : E[**F-ELEM**], **F-ELEM** -> E[**F-ELEM**]
adjonction d'un élément dans un ensemble

appe : E[**F-ELEM**], **F-ELEM** -> **BOOL**
appartenance d'un élément à un ensemble

e : **F-ELEM, +** -> E[**F-ELEM**]
construction d'un ensemble

elemsingl : E[**F-ELEM**] -> **F-ELEM**
élément d'un singleton

eq-e : E[**F-ELEM**], E[**F-ELEM**] -> **BOOL**
égalité pour les ensembles

evide : -> E[**F-ELEM**]
ensemble vide

evide? : E[**F-ELEM**] -> **BOOL**
un ensemble est-il vide ?

inclu : E[**F-ELEM**], E[**F-ELEM**] -> **BOOL**
inclusion d'un ensemble dans un autre

indef-e : -> E[**F-ELEM**]
indéfini pour l'ensemble

intersec : E[**F-ELEM**], E[**F-ELEM**] -> E[**F-ELEM**]
intersection de deux ensembles

single : **F-ELEM** -> E[**F-ELEM**]
création d'un ensemble à partir d'un élément

supe : E[**F-ELEM**], **F-ELEM** -> E[**F-ELEM**]
suppression d'un élément d'un ensemble

taillee : E[**F-ELEM**] -> **ENTIER**
taille (nombre d'éléments d'un ensemble)

union : E[**F-ELEM**], E[**F-ELEM**] -> E[**F-ELEM**]
union de deux ensembles

3.2.7 Opérations associées à S

acc : S[**F-ELEM**], **ENTIER -> F-ELEM**
accès dans une suite a un élément par son rang

ajout : S[**F-ELEM**], **F-ELEM -> S[F-ELEM**]
adjonction en tête d'une suite

ajoutrg : S[**F-ELEM**], **ENTIER, F-ELEM -> S[F-ELEM**]
insertion d'un élément dans une suite à un rang donné

app : S[**F-ELEM**], **F-ELEM -> BOOL**
appartenance d'un élément donné à une suite

conc : S[**F-ELEM**], S[**F-ELEM**] -> S[**F-ELEM**]
concaténation de deux suites

eq-s : S[**F-ELEM**], S[**F-ELEM**] -> **BOOL**
égalité pour les suites

indef-s : -> S[**F-ELEM**]
indéfini pour la suite

modifelt : S[**F-ELEM**], **ENTIER, F-ELEM -> S[F-ELEM**]
remplacement dans une suite d'un élément dont le rang est donné

s : **F-ELEM**], + -> S[**F-ELEM**]
construction d'une suite

singl : **F-ELEM -> S[F-ELEM**]
création d'une suite à partir d'un élément

sup : S[**F-ELEM**], **ENTIER -> S[F-ELEM**]
suppression dans une suite d'un élément donné par son rang

supelt : S[**F-ELEM**], **F-ELEM -> S[F-ELEM**]
suppression dans une suite de toutes les occurences d'un élément

svide : -> S[**F-ELEM**]
suite vide

svide? : S[**F-ELEM**] -> **BOOL**
une suite est-elle vide ?

rang : S[**F-ELEM**], **F-ELEM -> ENTIER**
rang ou place d'un élément dans une suite

taille : S[**F-ELEM**] -> **ENTIER**
taille (nombre d'éléments d'une suite)

tête : S[**F-ELEM**] -> **F-ELEM**
accès au premier élément (dernier ajouté en date) d'une suite

3.2.8 Opérations associées à T

acct : T[**F-ELEM, F-ELEM1**], **F-ELEM** -> **F-ELEM1**
accès dans une table à un élément donné par son indice

appt : T[**F-ELEM, F-ELEM1**], **F-ELEM** -> **BOOL**
appartenance d'un indice donné dans une table

eq-t : T[**F-ELEM, F-ELEM1**], T[**F-ELEM, F-ELEM1**] -> **BOOL**
égalité de deux tables

indef-t : -> T[**F-ELEM, F-ELEM1**]
indéfini pour la table

insert : T[**F-ELEM, F-ELEM1**], **F-ELEM, F-ELEM1**
-> T[**F-ELEM, F-ELEM1**]
insertion d'un élément dans une table

mod : T[**F-ELEM, F-ELEM1**], **F-ELEM, F-ELEM1**
-> T[**F-ELEM, F-ELEM1**]
modification d'un élément d'une table

supt : T[**F-ELEM, F-ELEM1**], **F-ELEM** -> T[**F-ELEM, F-ELEM1**]
suppression dans une table d'un élément donné par son indice

t : PC[**F-ELEM, F-ELEM1**], + -> T[**F-ELEM, F-ELEM1**]
construction d'une table

taillet : T[**F-ELEM, F-ELEM1**] -> **ENTIER**
taille (nombre d'indices accessibles d'une table)

tvide : -> T[**F-ELEM, F-ELEM1**]
table vide

tvide? : T[**F-ELEM, F-ELEM1**] -> **BOOL**
la table est-elle vide ?

3.2.9 Opérations associées à U

choisir : U[**F-ELEM, +**], **F-ELEM1, ENTIER -> U[F-ELEM, +]**
constructeur du type union
(peut-être utilisée plus agréablement par : CHOISIR-NOMTYPE (type))

défaire : U[**F-ELEM, +**], **ENTIER -> F-ELEM1**
donne une partie d'une union
(peut-être utilisée plus agréablement par :
DEFAIRE-NOMTYPE (union))

eq-u : U[**F-ELEM, +**], U[**F-ELEM, +**] -> **BOOL**
égalité de deux unions

est-type : U[**F-ELEM, +**], **ENTIER -> BOOL**
permet de savoir de quel type est une union
(peut-être utilisée plus agréablement par :
EST-TYPE-NOMTYPE (union))

indef-u : -> U[**F-ELEM, +**]
indéfini pour l'union

uvide : -> U[**F-ELEM, +**]
union vide

Annexe 4

Cette annexe reprend la syntaxe B.N.F. du méta-langage composé du langage de description d'objets et de relations inter-objets et du langage graphique.

<texte description> ::= <section objets> <section relations>
<section definitions-relations>
<section composition>
<section representation-relations>

<section objets> ::= **'Section OBJETS'** <suite objets>
<suite objets> ::= <objet> {<objet>}
<objet> ::= <type de noeud> **'represente_par'**
<concept graphique>
<section relations> ::= **'Section RELATIONS'**
<suite relations>
<suite relations> ::= <relation> {<relation>}
<relation> ::= **'relation'** <nom relation>
<suite sous-relations>
<suite sous-relations> ::= <sous-relation> {<sous-relation>}
<sous-relation> ::= <type de noeud> '('
<nom sous-relation> ')'
<type de noeud>
<section definitions-relations> ::= <definition relation>
{<definition relation>}
<definition relation> ::= **'relation'** <nom sous-relation>
'definie_par' <corps>
<section composition> ::= **'cas depart'** <suite regles>
'cas general' <suite regles>
['**cas exception**' <suite regles>]
<suite regles> ::= <regle> {<regle>}
<regle> ::= **'regle'** <expression composition>
<section representation-relations> ::= <contrainte> | <suite repr-composition>
<suite repr-composition> ::= <repr-composition> {<repr-composition>}
<repr-composition> ::= <mode composition> '.' <nombre entier>
'represente_par' <representation>
<representation> ::= <concept graphique> | **'RIEN'** | **'COLLE'**
<mode composition> ::= E|O|N|S|N-E|N-O|S-E|S-O|I|EES|EOS
<nombre entier> ::= <chiffre> {<chiffre>}
<chiffre> ::= 0|1|2|3|4|5|6|7|8|9
<expression composition> ::= <type de noeud> '[' <mode composition>
'.' <nombre entier> ']' <type de noeud>
<type de noeud> ::= <identificateur> [<nivocc>]

<nivocc> ::= <niv> | <niv> <occ>
<niv> ::= '**niv**' <id>
<occ> ::= '**occ**' <id>
<id> ::= <identificateur> |
<identificateur> '+' <nombre entier> |
<nombre entier>
<identificateur> ::= <lettre> {<lettre>}
<lettre> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U
|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q
|r|s|t|u|v|w|x|y|z
<concept graphique> ::= <id graphique> | <id graphique> '('
<expression graphique> ')'
<id graphique> ::= membre de la colonne nom-concept de la T.C.G. (table des concepts graphiques)
<expression graphique> ::= <expr graph> {<expr comp>}
<expr comp> ::= <mode composition> <expr graph>
<expr graph> ::= <operation sacso> | <chaine>
<nom relation> ::= <identificateur>
<nom sous-relation> ::= <identificateur>
<corps> ::= <suite instruction>
<suite instruction> ::= <instruction> {<instruction>}
<instruction> ::= <affectation> | <conditionnelle>
| <iteration> | <expression obt>
<affectation> ::= <variable> '**:=**' <expression>
<expression> ::= <operation sacso>
<expression obt> ::= <type de noeud> '**obtenu_par**'
<operation sacso>
<contrainte> ::= '**contrainte**' <identificateur> '('
<type de noeud simple> '**debut**'
<corps> '**fin**'
<type de noeud simple> ::= <identificateur>
<chaine> ::= '"' <lettre> {<lettre>} '"'
<vide> ::=
<variable> ::= <identificateur>
<operation sacso> ::= <op sacso> ['[' <suite arguments> ']']
<op sacso> ::= cfr annexe 3
<suite arguments> ::= <argument> {<argument>}
<argument> ::= <variable> | <nombre entier> | <type de noeud> |
<operation sacso>
<conditionnelle> ::= '**si**' <condition>
'**alors**' <suite instruction> |
'**si**' <condition> '**alors**' <suite instruction>
'**sinon**' <suite instruction>
<condition> ::= <expression simple> |
<expression simple>
<operateur relationnel>
<expression simple> | <operation sacso>

<operateur relationnel> ::= =|<|>|<>|<=|>=
<expression simple> ::= <terme> | <signe> <terme> |
<expression simple>
<operateur additif> <terme>
<terme> ::= <facteur> | <terme>
<operateur multiplicatif> <facteur>
<facteur> ::= <variable> | <constante arithmetique>
| '(' <expression> ')'
<constante arithmetique> ::= <nombre> | <chaine> | <id de constante>
<id de constante> ::= <identificateur>
<signe> ::= + | -
<operateur additif> ::= + | - | or
<operateur multiplicatif> ::= * | " | " | DIV | MOD | AND
<iteration> ::= **'iter' 'pour'** <partie iterative> **'de'**
<partie iterative> <suite instruction>
| **'iter' 'tant que'** <condition>
<suite instruction>
<partie iterative> ::= <variable> | <operation sacso> |
<constante arithmetique>

Annexe 5

Nous présentons quelques exemples de structures intermédiaires et leur représentation externe graphique.
Une boîte composition est représentée par une élipse, une inter-boîte et une boîte élémentaire par un rectangle. Pour chacune de ces boîtes, nous donnons également les valeurs de quelques champs spécifiques.
Pour une boîte graphique (fig. A5.1), nous donnons les champs suivants :

Figure A5.1 : champs représentés pour une boîte graphique

Pour une inter-boite (fig. A5.2), nous donnons les champs suivants :

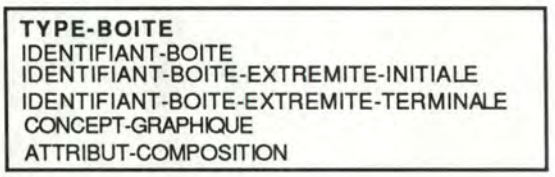


Figure A5.2 : champs représentés pour une inter-boîte

Pour un boîte élémentaire (fig. A5.3), nous donnons les champs suivants :

Figure A5.3 : champs représentés pour une boîte élémentaire

Certaines valeurs sont données en abréviation, à savoir BG pour boîte graphique, BT pour boîte textuelle, BE pour boîte élémentaire, IB pour inter-boîte, RECT pour rectangle, POLYG pour polygone.
Nous présentons en premier lieu la représentation externe graphique suivie de la structure intermédiaire graphique la décrivant.

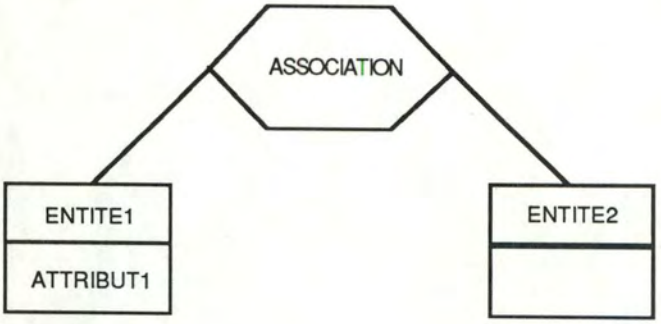


Figure A5.4 : représentation externe du premier exemple

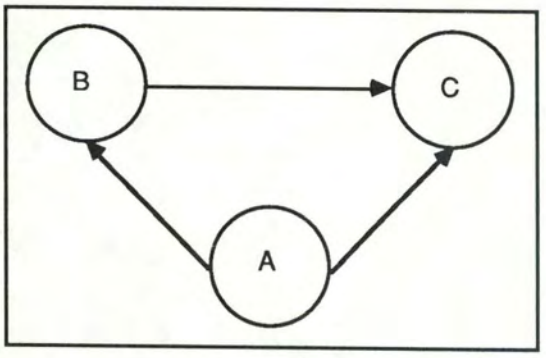


Figure A5.5 : représentation externe du deuxième exemple

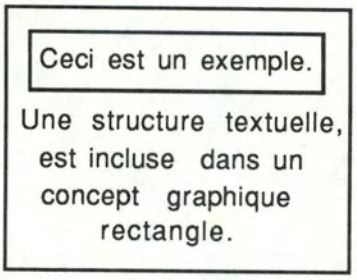


Figure A5.6 : représentation esterne du troisième exemple

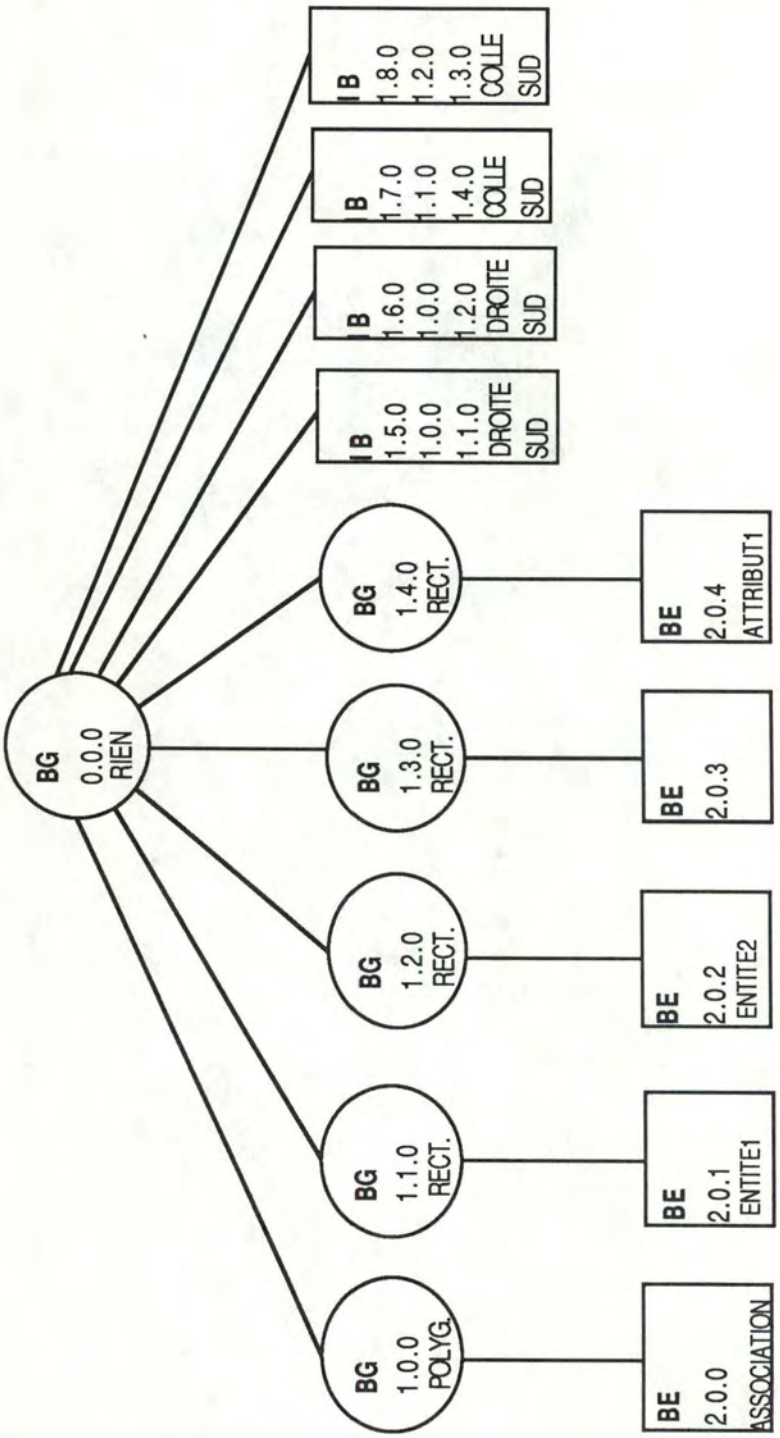


Figure A5.7 : S.I.G. du premier exemple

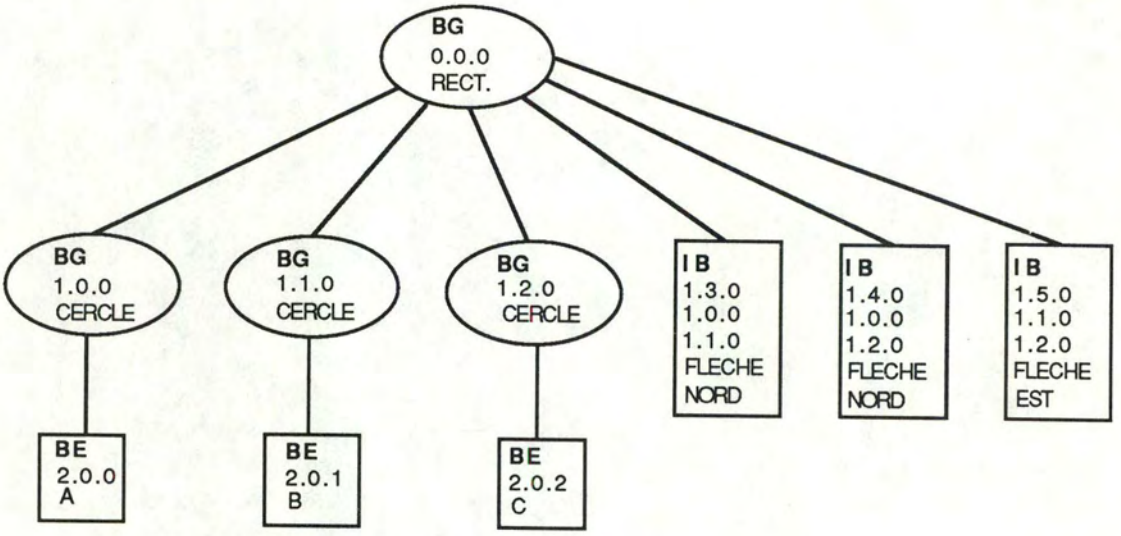


Figure A5.8 : S.I.G. du deuxième exemple

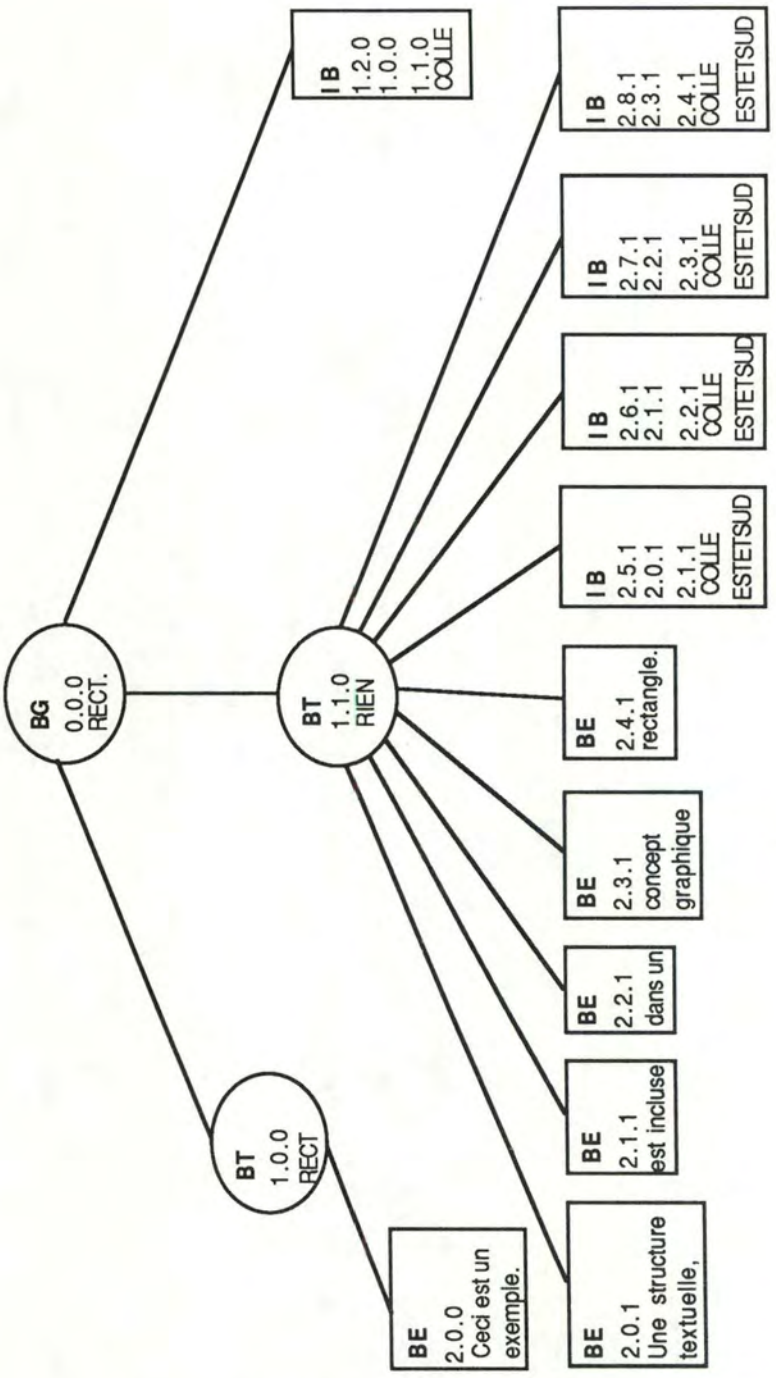


Figure A5.9 : S.I.G. du troisième exemple

REPARTITION DU TRAVAIL
DE CONCEPTION
ET DE REDACTION

REPARTITION

Nous reprenons dans cette partie, la répartition du travail de conception et de rédaction. La signification des abréviations utilisées est la suivante :

- /R pour rédaction,
- /C pour conception,
- MP pour Marc Paring,
- CP pour Christophe Paquet.

Introduction

Partie I CONTEXTE ET ENVIRONNEMENT DE TRAVAIL

Chapitre 1 SACSO : un Système d'Aide à la Conception de SpécificatiOns (MP/R)

Chapitre 2 X Window System : un outil de construction d'interfaces homme-machine (MP/R)

Chapitre 3 CEYX : un environnement de programmation générique et un langage orienté-objet (CP/R)

Conclusion

Bibliographie

Annexes